VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

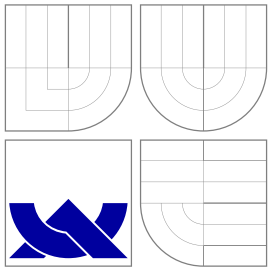
EVOLUČNÍ NÁVRH S VYUŽITÍM PŘEPISOVACÍCH SYSTÉMŮ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

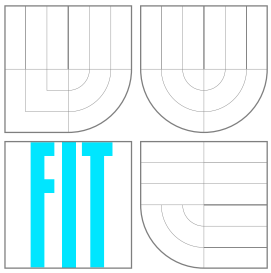
AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ HÝSEK

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EVOLUČNÍ NÁVRH S VYUŽITÍM PŘEPISOVACÍCH SYSTÉMŮ

EVOLUTIONARY DESIGN USING REWRITING SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ HÝSEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL BIDLO

BRNO 2007

Evoluční návrh s využitím přepisovacích systémů

Evolutionary Design Using Rewriting Systems

Vedoucí:
Bidlo Michal, Ing., UPSY FIT VUT

Zadání:

1. Seznamte se s problematikou různých typů přepisovacích systémů.
2. Seznamte se s problematikou evolučních algoritmů.
3. Navrhněte vhodnou reprezentaci a interpretaci přepisovacích pravidel (v závislosti na jejich typu).
4. Zvolte vhodnou doménu k demonstraci evolučního návrhu s využitím vybraných typů přepisovacích systémů.
5. Implementujte evoluční algoritmus pro nalezení přepisovacích pravidel k řešení Vámi zvoleného problému.
6. Diskutujte vhodnost různých typů přepisovacích systémů k řešení vybraných úloh.
7. Zhodnoťte dosažené výsledky.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty
informačních technologií Vysokého učení technického v Brně.

Abstrakt

Tato práce se zaměřuje na problematiku evolučního návrhu, věnuje se zejména problému zakódování kandidátního řešení. Běžně používané techniky evolučního návrhu pracují s kódováním kandidátního řešení, které není vhodné pro návrh rozsáhlých struktur. Práce se zabývá možným řešením popisovaného problému, tedy netriviálním převodem fenotypu na genotyp – developmentem. Tuto techniku demonstrujeme na evolučním návrhu posloupnosti přepisovacích pravidel umožňující konstrukci libovolně velkých řadicích sítí.

Klíčová slova

Evoluční návrh, Genetický algoritmus, Development, Přepisovací systém, Řadicí síť.

Abstract

This work provides an introduction to an evolutionary algorithms and evolutionary design. It also describes disadvantages of direct encoding of a genotype to phenotype and a method of nontrivial encoding which solves these problems. We are particularly talking about nonscalability of evolved solutions. We discuss a possible solution of described problem, nontrivial fenotype encoding called development. This technique is demonstrated on an evolutionary design of a sequence of rewriting rules which is able to construct an arbitrarily large sorting network.

Keywords

Evolutionary design, Genethic algorithm, Development, Rewriting system, Sorting network.

Citace

Jiří Hýsek: Evoluční návrh s využitím přepisovacích systémů, diplomová práce, Brno, FIT VUT v Brně, 2007

Evoluční návrh s využitím přepisovacích systémů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Michala Bidla. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Hýsek
22. května 2007

Poděkování

Děkuji především vedoucímu práce Michalu Bidlovi za jeho připomínky, korekturu a odborné vedení.

Obsah

1 Úvod **2**

2 Evoluční algoritmy **3**
2.1 Princip činnosti EA . . . 4
2.2 Základní rozdělení . . . 5
2.3 Genetické algoritmy . . . 6
2.3.1 Volba kódování jedince a fitness funkce . . . 6
2.3.2 Výběr jedinců pro křížení . . . 6
2.3.3 Křížení . . . 7
2.3.4 Mutace . . . 7
2.3.5 Obnovení populace . . . 7
2.4 Gramatická evoluce . . . 8
2.4.1 Backus-Naurova forma . . . 8
2.4.2 Zakódování chromozomu . . . 9
2.4.3 Genetické operátory využívané v GE . . . 9
2.5 Evoluční návrh . . . 9
2.6 Souhrn výhod a nevýhod evolučních algoritmů . . . 10

3 Řadicí sítě **11**
3.1 Popis sítě . . . 11
3.2 Princip 0-1 . . . 12
3.3 Konstrukce řadicích sítí . . . 13

4 Development **15**
4.1 Biologický development . . . 15
4.2 Využití developmentu v evolučním návrhu . . . 16
4.2.1 Výpočetní development jako přepisovací systém . . . 16

5 Návrh evolučního algoritmu **17**
5.1 Analýza algoritmu . . . 17
5.1.1 Zakódování genotypu . . . 17
5.1.2 Fitness funkce . . . 18
5.1.3 Výběr jedinců . . . 18
5.1.4 Křížení . . . 19
5.1.5 Mutace . . . 19
5.1.6 Obnova populace . . . 19
5.2 Development . . . 19
5.2.1 Typy pravidel pro rozvoj sítě . . . 21

5.2.2 Použitá pravidla 22

6 Experimenty **25**
6.1 Pevně daná embrya 25
6.1.1 Experiment I 25
6.1.2 Experiment II 30
6.2 Evolvovaná embrya 33
6.2.1 Zakódování kandidátního řešení 33
6.2.2 Experimenty 34

7 Závěr **38**

Kapitola 1

Úvod

S rostoucím potenciálem výpočetní techniky se otvírají nové možnosti pro řadu oborů. Astronomové mohou zpracovat nesrovnatelně větší množství dat v mnohem nižších časech, je možné předvídat složité fyzikální děje včetně počasí na určité části území, konstruktéři mohou simulovat vlastnosti navržených součástek včetně jejich chování při různých provozních podmínkách dříve, než je vyrobí, dokonce se počítače pomalu uplatňují i v oblastech, kde je zapotřebí kreativního myšlení vysoce kvalifikovaných odborníků, jako je návrh algoritmů, elektrických a číslicových obvodů, strojních součástek, radiokomunikačních antén nebo i komplexnějších konstrukcí jako jsou celé trupy lodí či profily křídel letadel.

V kreativním myšlení počítače za člověkem sice značně pokulhávají, zato tuto ztrátu začínají čím dál více dohánět hrubou výpočetní silou. Ta sama o sobě ještě dlouho nebude pro spoustu účelů dostatečná, ale v kombinaci s vhodně navrženými algoritmy prohledávání prostoru možných řešení začíná ukazovat svou použitelnost a slibné vyhlídky do budoucna. Protože počítače k návrhu nevyžadují detailní znalosti o funkci a chování daného systému jako člověk, stačí jim jen minimum informací k tomu, aby byly řešení schopny nalézt. V těchto doménách problémů se dnes uplatňují především evoluční algoritmy. Již dnes dokáží řešit problémy na úrovni kvalifikovaného člověka, o čemž svědčí např. každoroční udělování cen v mezinárodní soutěži human-competitive awards, které se udělují za řešení vyvinutá počítačem evolučním způsobem, která jsou shodná nebo lepší než nejlepší člověkem navržené řešení[1].

V této práci se budeme zabývat evolučním návrhem libovolně velkých řadicích sítí v využitím vývojového modelu založeného na přepisovacích systémech. První kapitola se zabývá problematikou evolučních algoritmů. Definujeme si v ní základní pojmy a vysvětlíme jejich princip. Obsahuje také podkapitolu o evolučním návrhu, jako jedno z možných využití evolučních algoritmů. Zaměřuje se hlavně na rozdíly mezi způsoby zakódování řešení, popisuje výhody složitějších zakódování pro evoluční návrh. Následuje popis řadících sítí, z čeho se skládají, jaké mají vlastnosti a jaké existují konvenční techniky návrhu řadících sítí pro libovolný počet vstupů.

[1] Jsou přesně daná kritéria pro uznání výsledku jako human-competitive. Je jich celkem 8, podrobější informace naleznete na URL http://www.genetic-programming.com/humancompetitivedefinition.html.

Kapitola 2

Evoluční algoritmy

Evoluce je princip, který je starší než lidstvo samo a přesto se začal výrazněji používat relativně nedávno. Historie využívání evolučních algoritmů sahá do padesátých let, ovšem do širšího povědomí vešly jako optimalizační metoda díky práci Johna Hollanda v první polovině sedmdesátých let. Jeho kniha [1] položila základ genetickým algoritmům, které byly později rozvíjeny a využívány na řešení řady různých problémů. Další důležitou osobností je David Goldberg, jehož kniha [2] se zabývá genetickými algoritmy z technického pohledu a snaží se je chápat jako algoritmy pro řešení široké řady úloh. Velkým přínosem byla také kniha Zbigniewa Michalewicze [3], který je autorem výrazných modifikací genetického algoritmu. V tomto stručném historickém přehledu také nelze opominout zakladatele genetického programování Johna Kozu a jeho knihy [4] a [5].

Evoluční algoritmy (EA) reprezentují množinu biologií inspirovaných algoritmů, které mají společný základ – využívají hledání nejlepšího řešení *přirozeným výběrem*. Přirozený výběr je klíčový proces evoluce a zajišťuje, že se prosadí ti nejsilnější. Slabší se buď přizpůsobí nebo vymřou. Další důležitý efekt evoluce je *genetický drift*, což se projevuje tím, že z genotypu mizí neúspěšné geny, vhodné vlastnosti postupně převládají a v konkurenčním boji o přežití se dále zlepšují. Třetím hlavním rysem je *proces reprodukce* – nová generace jedinců vzniká z původní populace, po které dědí část genetického materiálu.

Tyto principy samozřejmě nefungují pouze v přírodě, lze je analogicky aplikovat i na řešení nejrůznějších technických problémů. S výhodou se používají u těch, na které nestačí dnešní matematický aparát, výpočetní síla, nebo problém dostatečně neznáme a nedokážeme jej analyticky řešit.

Evoluční algoritmy se využívají k hledání nejsilnějšího jedince (reprezentujícího řešení daného problému), který je pokud možno co nejblíže optimálnímu řešení. Začaly se využívat zejména jako optimalizační metody, dnes jsou však využívány pro řešení problémů nejen optimalizačního charakteru. Nezanedbatelnou roli u nich hraje náhoda, ovšem narozdíl od stochastických optimalizačních metod, jako je Monte Carlo nebo např. simulované žíhání, je celý proces řízen deterministickými procesy, jako je např. výběr jedinců pro křížení nebo postup do další generace. Tím získává řadu výhod. Je možné provádět opatření proti uváznutí v lokálním extrému (např. pomocí přidávání určitého počtu nových náhodných jedinců do každé generace, či řízenou pravděpodobností mutace v závislosti na "síle" jedince), nastavením parametrů ovlivňovat a tím pádem i zvyšovat rychlost konvergence evoluce, apod.

Úspěšně se využily např. pro optimalizaci parametrů proudového motoru Boeingu 777 [7], kde úspora paliva v důsledku této optimalizace činila téměř 2.5%, což se projeví v obrovské úspoře na provozu letadel. Z úloh, které svou povahou nejsou přímo optimalizační, je možné EA využívat například pro návrh nejrůznějších věcí od nábytku po autonomní entity plnící

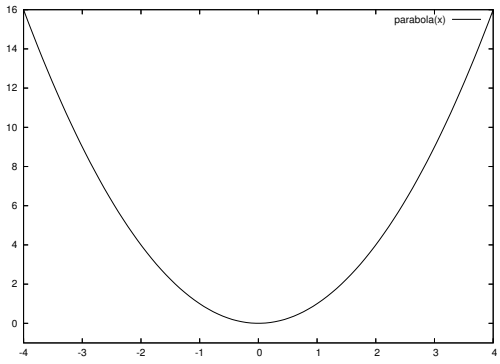
konkrétní úkoly v daném prostředí, hraní her nebo i generování hudby či obrazů. Spektrum problémů řešitelných těmito metodami je tedy velmi široké.

2.1 Princip činnosti EA

Nejprve si stručně popišme několik základních termínů, se kterými se pracuje v následujících kapitolách.

- *Jedinec* – jedno z kandidátních řešení v populaci.
- *Populace* – množina kandidátních řešení, které se v evolučním procesu vyvíjejí.
- *Fitness* – funkce udávající kvalitu jedince, tedy míru úspěšnosti řešení daného problému.
- *Genotyp* – zakódované kandidátní řešení. Řešení se kódují, aby se s nimi lépe pracovalo. Může to být např. řetězec znaků 1 a 0.
- *Gen* – jeden prvek genotypu. Např. je-li genotypem řetězec znaků 1 a 0, genem je každý z posloupnosti těchto znaků.
- *Fenotyp* – výsledné řešení, které vznikne po dekódování genotypu. Může to být např. posloupnost měst u problému obchodního cestujícího, obraz, profil křídla ledadla apod.
- *Reprodukce* – aplikace genetických operátorů na rodičovské genotypy za účelem získání potomků.
- *Diverzita populace* – rozmanitost genetického materiálu. Pokud má celá populace malou diverzitu (tzn. všichni jedinci se shlukuji blízko sebe v prohledávaném prostoru) hrozí větší nebezpečí, že uvázne v lokálním extrému.

EA v principu slouží k prohledávání n-dimenzionálního prostoru možných řešení. Můžeme si to představit jako přesouvání jedinců v prostoru tak, aby se místo, ve kterém se nacházejí, vyznačovalo co nejlepší cílovou vlastností. Kvalita nebo vhodnost polohy v prostoru řešení je pak udávána *fitness* funkcí. Evolucí se pak hledá extrém této funkce.

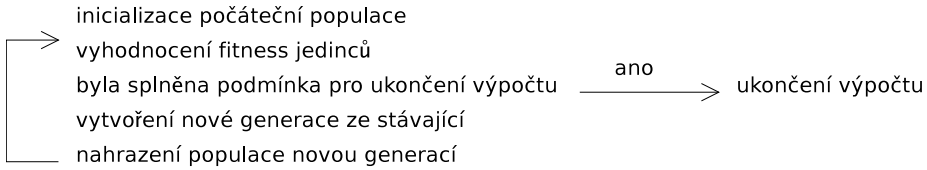


Obrázek 1: Ukázka jednoduchého 2D prostoru

Na Obr. 1 vidíme 2D prostor, kandidátní řešení jsou body na parabole, hledaný globální extrém je na pozici 0 na ose X. Hodnota funkce v tomto extrému je 0. Abychom tento bod evolucí našli, musí být hodnota fitness funkce pro tento bod nejvyšší. Tedy zde by byla $f(x) = -(x^2)$.

Zde extrém vidíme na první pohled sami, což ale u složitějších funkcích nebývá pravidlem, nehledě na to, že prostor řešení spousty problémů má větší množství dimenzí, než si dokážeme geometricky představit. Může být navíc velmi členitý a rozsáhlý, nebo ho dokonce nedokážeme popsat a zde už selhávají i tradiční metody s využitím dnešních nejvýkonějších počítačů.

Jak evoluční algoritmy při hledání řešení postupují? Obr. 2 ukazuje obecný průběh výpočtu v evolučních algoritmech.



Obrázek 2: Obecný cyklus výpočtu v evolučních algoritmech

Pracuje se se množinou možných řešení, každé z nich má přiřazenu hodnotu fitness. Dokud není splněna podmínka pro ukončení výpočtu, vytvářejí se další generace řešení. Přirozený výběr a genetický drift zajišťuje, že se v populaci jednotlivá řešení zlepšují. Jedinci s vyšší fitness mají větší šanci stát se rodiči a vyprodukovat tak více potomků, které mohou zdědit jejich kvalitní genetický materiál. Podmínkou pro ukončení výpočtu bývá nejčastěji nalezení řešení o dostačující kvalitě, nebo dosáhnutí maximálního počtu generací.

2.2 Základní rozdělení

Evoluční algoritmy se dají rozdělit do několika základních skupin, které se liší v povaze problému, pro jejichž řešení se využívají, a v implementačních záležitostech.

Genetické algoritmy [2] jsou nejpopulárnější podskupinou evolučních algoritmů, vyznačují se způsobem zakódování jedince. Říká se mu chromozóm a je to řešení zakódované do řetězce znaků 0 a 1. Tyto algoritmy se většinou používají na řešení optimalizačních problémů.

Genetické programování [4] se využívá k návrhu algoritmů nebo programů. Populaci tvoří velké množství jedinců, kteří jsou často ve formě stromu. Jejich fitness je schopnost řešit požadovaný problém.

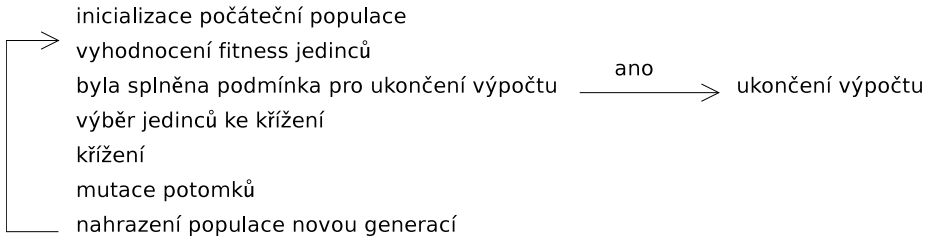
Evoluční strategie [10] pracuje s kandidátními řešeními jako s vektory reálných čísel. Používá se také pro optimalizační úlohy.

Evoluční programování [9] se využívá k návrhu programů, avšak jedinci jsou reprezentováni konečnými automaty. Jedinci se nekříží, jde o vývoj soupeřících druhů. Využívá se pouze mutace. Z každého rodiče se mutací vytvoří pouze jeden potomek.

To byl pouze výčet kategorií, se kterými je možné se setkat nejčastěji. Evoluční princip se dá využít, a také se to dělá, v rozličných skupinách algoritmů pro řešení nemalé skupiny problémů.

2.3 Genetické algoritmy

Genetický algoritmus je v současnosti nejčastěji používaným evolučním optimalizačním algoritmem. Hlavní rysy genetického algoritmu jsou způsob zakódování jedince a generování potomků pomocí křížení a mutace jedinců. Typický cyklus výpočtu v genetických algoritmech vidíme na obrázku 3.



Obrázek 3: Typický cyklus výpočtu genetického algoritmu.

2.3.1 Volba kódování jedince a fitness funkce

Volba fitness funkce a zakódování kandidátního řešení jsou klíčové části návrhu evolučního řešení problému. Záleží u nich na konkrétním řešeném problému, vyžadují jeho alespoň částečnou znalost. Při nevhodně zvolené fitness funkci může evoluce konvergovat velmi pomalu nebo výpočet může být zbytečně náročný. Genetický algoritmus (GA) využívá binární kódování – genotyp je řetězec nul a jedniček. U GA se jedinci říká *chromozóm.* Takto zakódovaná řešení se např. velmi snadno mutují a kříží. Genotyp je v tomto případě velmi jednoduchý, narozdíl od např. gentypů používaných u genetického programování, kde se jednotlivá kandidátní řešení reprezentují stromy, křížení pak znamená výměna podstromů mezi rodiči.

2.3.2 Výběr jedinců pro křížení

Aby se řešení zlepšovalo, musí mít větší šanci na křížení lepší jedinci, ovšem aby populace nezdegenerovala (neuvázla v nevýhodném lokálním extrému) musí mít šanci i ti horší, aby byla diverzita populace větší. Pro výběr jedinců pro křížení existuje řada strategií. Uveďme např. *turnajový výběr, ruletové kolo* nebo *pravděpodobnostní turnaj.*

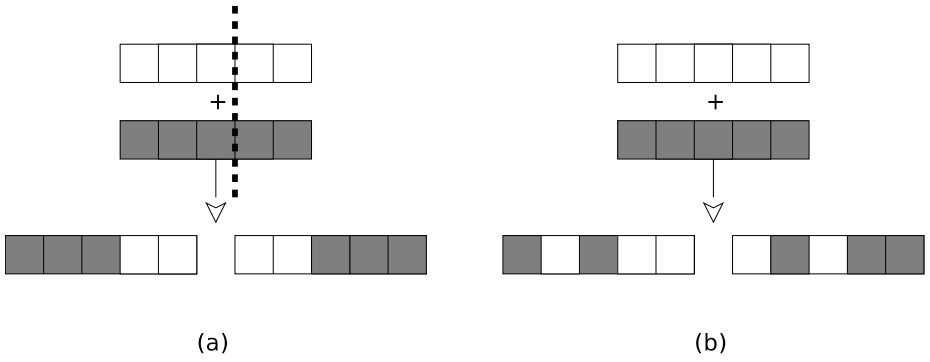
Turnajový výber je založený na soupeření jedinců o možnost křížení. Z populace se vyberou náhodně 2 jedinci, ten s lepší hodnotou fitness postupuje do křížení. To splňuje podmínku, že lepší mají větší šanci na úspěch, zároveň tak díky náhodnému výběru jedinců zachovává diverzitu populace. Pokud se vyberou dva jedinci představující nekvalitní řešení, jeden z nich bude křížen a tedy jeho nekvalitní geny přejdou do další generace. Což nemusí vadit, protože po křížení s jiným jedince může vzniknout genotyp, který bude výrazně lepší. A i kdyby ne, tyto geny předcházejí situaci, kdy celá populace uvázne v lokálním extrému.

Ruletové kolo je strategie výběru jedince s pravděpodobností určenou jeho fitness. Jsou tedy opět vybíráni náhodní jedinci z pomyslného ruletového kola, ovšem čím větší má jedinec fitness, tím větší část ruletového kola mu náleží, a má tedy větší šanci na výběr.

Pravděpodobnostní turnaj je kombinace předchozích dvou strategií. Z „ruletového kola" se vyberou 2 jedinci a ten lepší z nich je vybrán pro křížení. Tato strategie má větší *selekční tlak* než předchozí dvě. Selekčním tlakem rozumíme schopnost strategie výběru zvýhodňovat kvalitní jedince a naopak ty méně kvalitní potlačovat.

2.3.3 Křížení

- jednobodové – náhodně se vybere pozice (bod křížení), v níž se rodičovské genotypy rozdělí na dvě části. Levá část jednoho rodiče se spojí s pravou druhého a naopak. Tím vzniknou 2 potomci. Na Obr. 4 je příklad vzniku jednoho potomka s náhodným bodem křížení (na obr. vyznačen přerušovanou čarou) .
- dvoubodové – analogicky k jednobodovému, pouze s tím rozdílem, že body křížení jsou 2
- uniformní – každý gen potomka je náhodně vybrán z jednoho nebo z druhého rodiče [11]



Obrázek 3: Příklady druhů křížení (a) jednobodového (b) uniformního.

2.3.4 Mutace

Mutace je náhodná změna některých genů. Probíhá s určitou malou pravděpodobností a jejím účelem je zvyšovat diverzitu populace. Pokud řešení uvázne v lokálním extrému a většina jedinců v populaci si je velmi podobná, pak křížení nepomůže k opuštění tohoto místa. Ovšem pokud se některému jedinci změní náhodně gen nebo i více, zanese se tím do populace nový genetický materiál, křížením se rozšíří do větší části populace a to jí jako celku dává možnost dále prohledávat i jiné části prostoru, než onen lokální extrém. Mutace je tedy velmi důležitá. Diverzita populace se však dá zajistit i jinými způsoby, jako přidáváním určitého počtu náhodně vygenerovaných jedinců do nové generace.

Některé algoritmy (např. umělé imunitní systémy) nepoužívají křížení, ale pouze mutaci. V takovém případě jde většinou o *hypermutaci*. Nemusí totiž nastat s malou pravděpodobností, ale může celý genotyp z větší části změnit. Jedinec se mutuje s pravděpodobností nepřímo úměrnou jeho fitness. To způsobí, že modifikace jedince je tím větší, čím je dál od hledaného místa. Je-li blízko řešení, prostor se prohledává po menších krocích.

2.3.5 Obnovení populace

Pro obnovení populace existují dvě hlavní strategie.

Úplná obnova, při níž všichni rodiče vymřou a populace je tak nahrazena potomky. Při této strategii se může stát, že žádný z potomků nebude dosahovat takové kvality, jako některý z rodičů, tudíž přijdeme o lepší řešení.

Částečná obnova, část původní populace zůstane zachována i v další generaci. Tato strategie zajišťuje, že nejlepší nalezené řešení v populaci se mění pouze směrem k lepšímu.

Jediný způsob, jak nejlepšího jedince z populace odstranit je jeho nahrazení kvalitnějším řešením. Způsobů výběru jedinců z původní populace pro přesun do další generace je několik. Potomci mohou nahradit např. jen určité procento populace tak, že nahrazují nejslabší jedince. Pokud je však většina potomků kvalitnějších než většina rodičů, můžeme přijít o kvalitní řešení. To se nestane např. při sjednocení rodičů i potomků a výběru nejlepších jedinců tak, abychom zachovali velikost populace. Nová generace se tak bude skládat z nejkvalitnějších rodičů i potomků. Tahle strategie velmi znevýhodňuje nekvalitní řešení, což má za následek rychlou konvergenci k extrému, ale také potenciálně nižší diverzitu populace. Obnovit populaci lze také turnajem. Sjednotí se rodiči a potomci a z nich se náhodně vybírají dva jedinci, z nichž lepší postoupí do další generace.

Důležitým pojmem při obnovování populace je *elitismus*. Je to strategie, při které jsou nejlepší řešení zachovávána v další generaci. Ve většině strategií pro částečnou obnovu je elitismus patrný.

2.4 Gramatická evoluce

Gramatická evoluce (GE) je jednou z nejnovějších evolučních technik. Její počátky sahají do roku 1998 [12]. Jde o kombinaci genetických algoritmů a genetického programování. Stejně jako u genetického programování je cílem GE automatický návrh programů. Je ovšem obecnější – pomocí GE je možné navrhovat programy v libovolném jazyce, který může být popsán bezkontextovou gramatikou, resp. Backus-Naurovou formou (BNF). Na rozdíl od genetického programování, jsou jedinci reprezentováni produkčními pravidly dané gramatiky. Pravidla jsou zakódována do binárního řetězce a používá se jednoduché jednobodové křížení, tedy rysy typické pro klasický genetický algoritmus. Výhodou je, že generované programy nejsou omezeny konkrétním programovacím jazykem a navíc takto reprezentovaní jedinci bývají narozdíl od stromové reprezentace výrazně menší [12].

2.4.1 Backus-Naurova forma

BNF je notace používaná k vyjádření bezkontextových gramatik. Je to tedy formální způsob popisu bezkontextového jazyka. Formálně může být bezkontextová gramatika definována jako čtveřice $\{N, T, P, S\}$, kde N je množina *neterminálů*, T je množina *terminálů*, P množina přepisovacích (produkčních, derivačních) pravidel a S značí počáteční neterminál. *Neterminál* je symbol, který je možné přepsat na *řetězec* terminálů a/nebo neterminálů. *Terminál* je symbol, který je součástí cílového jazyka, atomické symboly, které nelze dále přepisovat. Produkční pravidla mají tvar $V \rightarrow w$, kde V je neterminál a w je řetězec, na který je neterminál monžné přepsat.

BNF je množina produkčních (nebo také derivačních) pravidel. Pravidlo má následující tvar

neterminál ::= řetězec

Pokud existuje více pravidel pro přepis daného neterminálu, pravá strana pravidla obsahuje více řetězců oddělených znakem |. Např. BNF popis jednoduché gramatiky popisující aritmetické výrazy může vypadat následovně:

```
expr ::= op expr exp | var
op   ::= + | - | * | /
var  ::= X | Y
```

2.4.2 Zakódování chromozomu

Jak jsme se zmínili výše, genotyp v gramatické evoluci je stejně jako u genetických algoritmů binární řetězec a říká se mu chromozóm. V GE mají chromozomy proměnnou délku. Chromozómem je reprezentována posloupnost pravidel tak, jak budou postupně aplikována. Je dělen na osmibitové geny, které vyjadřují číslo pravidla. Protože je pravidel většinou méně než 256, je na toto číslo aplikována operace modulo počtu pravidel pro aktuálně rozvíjený neterminál.

Je zřejmé, že toto zakódování umožňuje reprezentaci jednoho pravidla několika různými řetězci. Máme-li např. k dispozici 5 pravidel, 2. pravidlo reprezentují geny s hodnotou 1, 6, 11, 16, atd. To umožňuje tzv. *tiché mutace*, kdy změny v chromozomu nemají vliv na výsledný program.

2.4.3 Genetické operátory využívané v GE

GE využívá všechny operátory genetického algoritmu, přidává však dva nové – *duplicate* (zdvojení) a *prune* (ořezání) [12].

Duplicate provádí prostou kopii genu. Nově vytvořený gen nemusí být přímo vedle svého vzoru, může být umístěn kdekoli v chromozomu. Operátor se využívá pro duplikaci více genů, jejich počet je zvolen náhodně. Duplikace genů přináší několik výhod. Genetický materiál je méně náchylný k destruktivním mutacím – pokud mutace znefunkční daný gen, jeho kopie jej nahradí. Umožňuje vytvářet nové, složitější fuknce, které vzniknou např. vhodnou mutací duplikovaného genu. Duplikace je také vhodná v případech, kdy je potřeba z kratších chromozomů vytvořit složitější řetězce, reprezentující komplexní struktury. Geny jsou v tomto případě stavební bloky, ze kterých se skládá výsledný řetězec.

Jedinci v GE nepotřebují nutně použít všechny jejich geny. Přebytečné geny jsou v evolučním procesu vhodné ke snížení pravděpodobnosti poškození klíčových genů během křížení či mutace. Ovšem toto nemusí v důsledku přinášet příliš užitku. Stejně tak křížením či mutací kvalitního genu můžeme také získat ještě kvalitnější jedince. Křížením neužitečných nebo nefunkčních genů mezi sebou nemusí mít žádný účinek a může výrazně prodloužit evoluční proces. Pro snížení počtu takových genů se používá operátor **prune**, který je aplikován s určitou pravděpodobností na všechny jedince obsahující geny, které se neuplatní při převodu genotypu na fenotyp. Operátor prune odstraní všechny tyto geny z daného jedince. Tím se dosáhne rychlejší evoluce a přínosnějšímu křížení [12].

2.5 Evoluční návrh

Evoluční návrh se zabývá konstrukcí složitějších struktur z jednoduchých stavebních bloků s využitím evoluce. Populární je evoluční návrh logických obvodů na programovatelných hradlových polích FPGA, nebo např. návrh virtuálních tvorů schopných vykonávat požadovanou činnost [13].

Evoluční algoritmy popsané v předchozí kapitole mají genotyp (kandidátní řešení) zakódován do struktury, jejíž velikost je přímo úměrná velikosti požadovaného fenotypu. Čím chceme mít komplexnější fenotyp, tím samořejmě musí být rozsáhlejší i genotyp. Z toho vyplývá několik problémů.

1) Se zvyšující se velikostí požadovaného fenotypu se zvyšuje i velikost genotypu a tím i množství dat, které je v evolučním procesu nutné zpracovat. Čím jsou genotypy větší, tím se samozřejmě zvětšuje prostor, ve kterém se řešení nachází, a velmi snadno se může stát, že

evoluční algoritmus nezvládne efektivně s dnešním dostupným výpočetním výkonem takový prostor prohledat.

2) Řešení nejsou škálovatelná. Pokud chceme řešení jiné velikosti, je potřeba celý evoluční proces opakovat znovu s genotypy odpovídajícího rozsahu.

Tyto problémy jsou v evolučním návrhu mnohem palčivější, než u běžných optimalizačních úloh, kde jde o nalezení správné kombinace číselných parametrů. Začalo se tedy experimentovat s jinými, netriviálními způsoby zakódování kandidátního řešení. Existuje několik přístupů, obecně jde však o využití genotypů, které neobsahují přímo zakódovaný fenotyp, ale instrukce nebo pravidla pro jeho vytvoření. Opakovaným prováděním těchto instrukcí získáváme rozsáhlejší fenotypy. Vyhodnocení fitness funkce jedince pak znamená výrobu fenotypu o požadované velikosti aplikací instrukcí obsažených v genotypu a výpočet jeho vhodnosti.

2.6 Souhrn výhod a nevýhod evolučních algoritmů

Hlavní výhodou EA je, že nevyžadují analytický popis systému, jsou schopny řešit problémy typu „černá skříňka". Důležité je dokázat vyjádřit míru správnosti daného řešení.

Vzhledem k tomu, že pracujeme s celou populací, je možné nalézt několik různých řešení v jednom běhu algoritmu. Řeší tedy i multikriteriální úlohy.

Evoluční algoritmy mají tendenci konvergovat ke globálnímu extrému. Opět díky populaci řešení, se kterými pracujeme. Uvázne-li část populace v lokálním extrému, je pravděpodobné, že v určitém čase zbytek populace objeví lepší řešení.

Této skupině algoritmů obvykle nedělá problém prohledávat multidimenzionální a členité prostory.

Nevýhodou evolučních algoritmů je často velká závislost rychlosti konvergence nebo schopnosti nalézt globální extrém na nastavení počátečních parametrů, případně na zakódování řešení. Evoluční algoritmy jsou také ve srovnání s běžnými optimalizačními metodami výpočetně náročnější. Např. gradientní metody nebo simulované žíhání pracují s jedním řešením, evoluční algoritmy s celou populací.

Kapitola 3

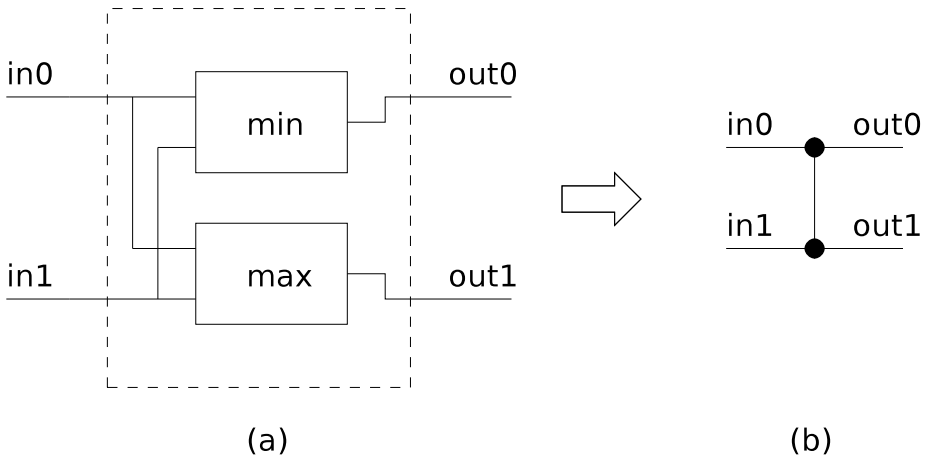
Řadicí sítě

Protože se v této práci budeme zabývat aplikací popisovaných principů na konkretní zvolený problém, kterým je návrh řadicích sítí, popišme si, jak vypadají a jaké jsou konvenční způsoby jejich konstrukce.

3.1 Popis sítě

Problematiku řadicích sítí shrnul a popsal D. Knuth v [15]. Řadicí síť je kombinační obvod s N vstupy a N výstupy. Jejím výstupem jsou vstupní hodnoty seřazené do neklesající posloupnosti.

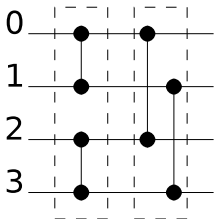
Základním blokem řadící sítě je *komparátor*. Má dva vstupy a dva výstupy. Jeho blokové schema vidíme na obr. 5. Ze schematu je patrná jeho funkce – menší hodnota z obou vstupů se předá na výstup *out0*, větší naopak na *out1*. Jde tedy o *compare-and-swap* operaci. U řadicích sítí se používá zjednodušená grafická reprezentace zobrazená na pravé části obr. 5 b).



Obrázek 5: Komparátor: (a) Blokové schema a (b) jeho zjednodušená reprezentace.

Vrstva komparátorů je posloupnost komparátorů, jejichž vstupy a výstupy jsou odlišné jak v rámci komparátoru, tak i v rámci celé vrstvy. Tedy žádné dva komparátory v dané vrstvě nejsou připojeny na stejný vstup ani výstup a každý komparátor je připojen na dva odlišné vstupy a má dva odlišné výstupy. To znamená, že komparátory ve vrstvě nemohou být vzájemně ovlivňovány, tudíž mohou pracovat paralelně.

Síť komparátorů je posloupnost vrstev komparátorů. Její délka, tedy počet komparátorových vrstev v ní obsažených, určuje *zpoždění sítě*. Je to počet kroků, které síť během řazení vykoná. Další důležitou vlastností sítě je její *velikost*. Jde o celkový počet komparátorů obsažených v síti. Příklad dvouvrstvé sítě komparátorů vidíme na obr. 6. Jsou zvýrazněny jednotlivé vrstvy. Zpoždění sítě je tedy 2, velikost sítě je 4.



Obrázek 6: Příklad dvouvrstvé sítě komparátorů.

Řadicí síť je síť komparátorů, pro kterou platí, že seřadí všechny kombinace vstupů do neklesající posloupnosti.

Sítě budeme reprezentovat i v textové podobě. Komparátor je vyjádřen dvojicí (a, b), kde a je index vstupu sítě, na který je připojen komparátor $in0$, počínaje nulou. b je relativní index vstupu, na který je připojen komparátor $in1$. Komparátor připojený na vstupy 1 a 3 má tedy textovou podobu $(1, 2)$. Síť komparátorů na obrázku 6 reprezentuje řetězec $(1, 1)(2, 1)(0, 2)(1, 2)$.

3.2 Princip 0-1

Pro ověření, zda daná síť skutečně správně řadí všechny kombinace vstupů, je nutné ověřit správnost pro n^n různých kombinací, což je obrovské množství, tento problém patří do třídy NP-úplných. Pro představu u 10 vstupé sítě mluvíme o deseti miliardách možných vstupů.

Naštěstí pro kontrolu správnosti sítě není díky tzv. *principu 0-1* (zero-one principle) nutné simulovat řazení pro všechny vstupy . Je dokázáno [15], že pokud síť správně seřadí všech 2^n kombinací čísel 0 a 1, pak správně řadí kombinace jakýchkoli čísel. Tím enormně zredukujeme počet nutných řazení. Např. u zmiňované desetivstupé sítě se tento počet z původních deseti miliard sníží na pouhých 1024.

Síť komparátorů na obr. 6 tedy není řadicí síť, protože neseřadí správně všech 16 kombinací čísel 0 a 1. V tabulce 1 si můžeme prohlédnout výstupní hodnoty, které síť vrátí, pro všechny zmiňované. Zvýrazněny jsou řádky se špatně seřazenými vstupy:

vstup	výstup	vstup	výstup
0000	0000	1000	0001
0001	0001	**1001**	**0101**
0010	0001	**1010**	**0101**
0011	0011	1011	0111
0100	0001	1100	0011
0101	**0101**	1101	0111
0110	**0101**	1110	0111
0111	0111	1111	1111
0111	0111	1111	1111
0111	0111	1111	1111
0111	0111	1111	1111
0111	0111	1111	1111
0111	0111	1111	1111
0111	0111	1111	1111

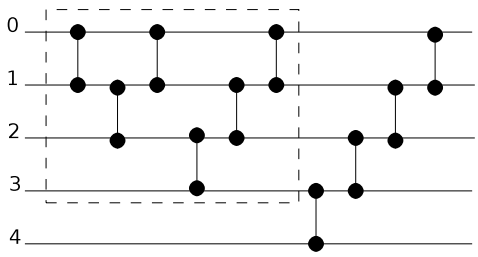
Tabulka 1: Výstupy sítě z obr. 6 pro všechny vstupní kombinace nul a jedniček.

3.3 Konstrukce řadicích sítí

Řadicí sítě jsou obvykle navrhovány pro předem daný pevný počet vstupů. Existuje však i několik konvenčních postupů konstrukce libovolně velkých řadících sítí. Jde o princip *vkládání*, který se využívá např. v algoritmu insert sort, a o princip *výběru*, který známe např. z algoritmu bubble sortu.

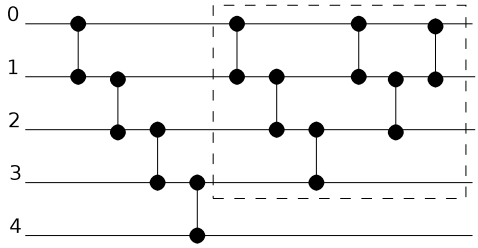
Při vkládání se každý $m + 1$. vstup vloží na správnou pozici do seřazené posloupnosti předchozích m vstupů. Tím se postupně vytváří seřazená posloupnost od nejnižších indexů vstupní datové sekvence.

Síť zkonstruujeme přidáním dalšího vstupu a posloupnosti komparátorů na konec sítě (za poslední komparátor) tak, že porovávají sousední vstupy postupně od vstupů s nejvyšším indexem až po první vstup. Všimněme si, že takto vyrobíme jakkoli velkou síť, včetně dvouvstupé. Ta je rozšířením jednovstupové, což je pouze jeden vodič bez komparátoru. Příklad konstrukce je zobrazen na obr. 7, který ukazuje konstrukci pětivstupé řadící síť, která vznikla rozšířením čtyřvstupé. Původní čtyřvstupá síť je na obr. zvýrazněna přerušovaným obélníkem a může jít o jakoukoli platnou řadicí síť. Přidáním vhodného uspořádání komparátorů z ní vznikne síť větší. Popsaný princip se dá aplikovat už od jednovstupé „řadicí sítě" (jeden vstup bez komparátoru) po síť s libovolně velkým počtem vstupů.



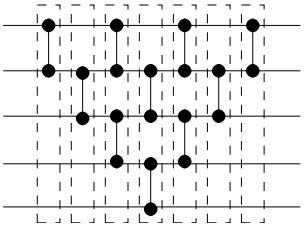
Obrázek 7: Příklad konstrukce 5-vstupové sítě ze 4-vstupové s využitím principu vkládání

U výběru se naopak vybírá nejvyšší prvek z neseřazené části a ukládá se na začátek seřazené posloupnosti, která se tvoří od nejvyšších indexů (tzn. na konci vstupního pole). Takovou řadící síť sestrojíme opět rozšířením menší sítě – přidáním dalšího vstupu a posloupnosti komparátorů na začátek sítě (před první komparátor) tak, že porovnávají sousední vstupy postupně od prvního až po poslední. Na obr. 8 je pětivstupá řadící síť vytvořená podle tohoto principu ze sítě čtyřvstupé.



Obrázek 8: Příklad konstrukce 5-vstupové sítě ze 4-vstupové s využitím principu výběru

Vidíme, že oba přístupy jsou velmi podobné. Řadicí sítě vypadají zrcadlově obrácené, jsou však identické. Jak víme, nekolidující sousední komparátory mohou pracovat paralelně. Po překreslení sítí sestrojených za pomocí obou principů do sekvence vrstev komparátorů, získáme stejnou síť, jak vidíme na obr. 9. Na obrázku jsou zvýrazněny jednotlivé vrstvy. V prostředních třech pracují 2 komparátory paralelně.



Obrázek 9: Paralelní vrstvy řadicí sítě

Při konstrukci řadicích sítí je kladen důraz na její velikost (počet použitých komparátorů) a na zpoždění sítě. Stejně jako bubble sort a insert sort nejsou ani zdaleka optimální řadící algoritmy, tak ani takto vytvořená síť není optimální. Obsahuje zbytečně velké množství komparátorů a má velké zpoždění. Efektivnější sítě se však navrhují speciálně pro konkrétní počet vstupů, nejsou tedy na rozdíl od takto vytvořených sítí šklálovatelné.

Kapitola 4

Development

Ve třetí kapitole jsme si řekli, že existuje několik přístupů k netriviálnímu mapováni genotypu na fenotyp, které se používají zejména kvůli škálovatelnosti navrženého řešení. V této práci se budeme zabývat pouze jedním z nich a to *developmentem*.

4.1 Biologický development

Development je postupný vývoj složitých organizovaných struktur z velmi jednoduché počáteční skupiny stavebních bloků. Jakým způsobem zajistit, aby se z minimální skupiny bloků nebo i z jediného bloku vyvinula komplexní struktura s požadovanými vlastnostmi? Při řešení tohoto problému se opět můžeme inspirovat biologickými procesy, v tomto případě vývojem embrya. Z několika buňek se dokáže utvořit přesně daná struktura nebývalého rozsahu, tvořící vzájemně spolupracující orgány, které tvoří samostatně fungující jedince.

Biologický development zahrnuje spoustu neobyčejně složitých procesů a prozatím ne všem lidstvo plně rozumí. Nám však dobře poslouží i základní rysy. Chování všech buňek je řízeno proteiny, které jsou produkovány geny. Proteiny zastávají spoustu funkcí, které mohou ovlivňovat buňky, jiné proteiny nebo i samotné geny. Nejdůležitější je pro nás vliv protenů na samotné buňky. Později si ukážeme analogii těchto procesů, kterou budeme při návrhu využívat.

Development je působení proteinů na z počátku malé množství buněk, kterému říkáme embryo. Díky tomuto působení se embryo vyvíjí až do konečné podoby, která je dána genotypem. Genotyp určuje tvar a funkci organismu, proces developmentu jej vytváří. Development se skládá z pěti hlavních procesů.

Buněčné dělení probíhá za účelem zvětšování vyvíjející se struktury. Dělení je rovnoměrné a žádná z buňek po dělení neroste, mají tedy obě stejnou velikost.

Formování tvaru buněčné struktury je proces, při kterém se buňky v rámci embrya přesouvají za účelem vzniku požadované organizované struktury.

Morfogeneze je proces, během kterého dochází k rozsáhlým přesunům buněk. Příkladem je formování vnitřních orgánů. Buňky na okraji embria migrují směrem dovnitř, kde se formují s ostatnímy a vytvářejí tak orgány.

Diferenciace buňek je postupný proces, díky kterému buňky dosahují odlišných struktur a funkcí, což má za následek vznik různých typů buněk, jako např. neurony, buňky kůže, apod. Diferenciaci způsobuje také asymetrické dělení buněk – každá z buněk má po dělení jinou velikost.

Růst znamená kromě množení buněk i zvyšování velikosti buněk.

Tyto procesy nemusí nastávat samostatně, mohou se vzájemně překrývat.

4.2 Využití developmentu v evolučním návrhu

V evolučním návrhu se většinou pracuje s genotypem, který je přímým zakódováním fenotypu. V kapitole 2.4 jsme si shrnuli nevýhody tohoto přístupu. Pro převod genotypu na fenotyp využijeme analogie k biologickému developmentu. Genotyp vytváří proteiny, které formují výslednou strukturu požadované složitosti z počátečního embrya. Zakódované kandidátní řešení je skupina proteinů, tedy řekněme předpis pro vývoj embrya. Jejich opakovanou aplikací dosáhneme řešení další instance daného problému – struktury stejného tvaru a funkce, ale větší velikosti (pokud obsahuje proteiny způsobující růst, což je samozřejmě žádané), např. řadící sítě pro větší počet vstupů. Chceme-li opět získat řešení další instance problému, vezmeme aktuální řešení jako embryo a aplikujeme na něj instrukce pro růst (proteiny) dané genotypem.

biologický development	výpočetní development
genotyp	předpis pro vytvoření kandidátního řešení
embryo	počáteční řešení (nemusí být kompletní)
development	netriviální převod genotypu na fenotyp
proteiny	instrukce pro přepis embrya na další instanci

Tabulka 2: Shrnutí analogie pojmů v biologickém developmentu a výpočetním

4.2.1 Výpočetní development jako přepisovací systém

Přepisovací systém by se dal definovat jako soubor přepisovacích pravidel k transformaci struktur (řetězců, grafů, výrazů, ...).

Výpočetní development je proces, v němž se opakovanou aplikací pravidel na strukturu vytváří řešení, které má stejnou funkci nebo účel jako počáteční, má však jinou velikost nebo složitost.

Navrhovaný objekt je reprezentován řetězcem symbolů. Genotypem je soubor pravidel přepisovacího systému. Aplikací pravidel na embryo, přepíšeme řetězec do nové formy, která bude odpovídat řešení další instance problému. Přepisovacím systémem můžeme rozumět např. gramatiku nebo její popis v BNF, jak je tomu u gramatické evoluce popisované v kapitole 2.4. Od developmentu se však liší tím, že pravidla aplikuje tak dlouho, dokud nevznikne řetězec terminálů, který reprezentuje výsledné řešení. Tento řetězec už není možné dále přepisovat.

Kapitola 5

Návrh evolučního algoritmu

5.1 Analýza algoritmu

Pro hledání vhodných přepisovacích pravidel byl využit upravený genetický algoritmus. V této kapitole si rozebereme jednotlivé vlastnosti použitého algoritmu a popíšeme důvody, proč je algoritmus navržen právě tímto způsobem.

5.1.1 Zakódování genotypu

Ještě než si vysvětlíme funkci použitého evolučního algoritmu, podívejme se, jak je kandidátní řešení reprezentováno.

Protože je řešeným problémem návrh libovolně velkých řadicích sítí pomocí developmentu a genotypem je tudíž soubor pravidel pro rozvoj embrya, je zakódování genotypu inspirováno gramatickou evolucí. Existuje tedy sada daných pravidel a cílem evoluce je najít jejich vhodnou kombinaci.

Genotypem je binární řetězec, který reprezentuje posloupnost pravidel. Každý gen má velikost 6 bitů. Dále existuje soubor očíslovaných pravidel. Gen reprezentuje pravidlo dané následujícím vztahem:

```
pravidlo = hodnota_genu mod počet_pravidel
```

Mějmě např. definovaných 15 pravidel a chromozom zobrazený na obrázku 10. Hodnoty jednotlivých genů jsou (3, 23, 57, 13, 4). Chromozom tedy reprezentuje sekvenci pravidel (3, 8, 12, 13, 4).

V programu využíváme 32 pravidel, přičemž ne všechny mají funkci. Některá pravidla jsou tzv. *NOP* (no operation) pravidla, která umožňují evolvovat chromozomy s proměnným množstvím pravidel, které mají vliv na rozvíjení embrya.

000010	010111	111001	001101	000100
3	23	57	13	4

Obrázek 10: Příklad chromozomu.

5.1.2 Fitness funkce

Pro zjištění míry kvality řešení je nutné zkonstruovat řadicí síť z embrya za použití pravidel v daném genotypu. Existuje několik kritérií, která určují kvalitu sítě. Nejduležitější je její funkčnost, tedy jestli funguje správně pro jakýkoli možný vstup. Dále požadujeme, aby daná síť byla škálovatelná, což je právě důvod, proč ji generujeme pomocí vyevolvovaných pravidel a neevolvujeme ji přímo. A v poslední řadě nás zajímají parametry sítě, jako je počet použitých komparátorů či její zpoždění.

Chování výsledné sítě se simuluje na všech 2^w vstupních kombinací čísel 0 a 1, což nám podle *zero-one principu* dostačuje pro určení, zda síť dokáže seřadit jakoukoli vstupní posloupnost do neklesající posloupnosti. Genotyp je tím kvalitnější, čím více vstupů dokáže jím zkonstruovaná síť správně seřadit. Rozdíl mezi sítí, která dokáže seřadit $2^w - 1$ vstupů a sítí řadící správně všechny vstupy je však významnější než v ostatních případech, proto se v případě správného seřazení všech vstupů přidává k fitness konstantní hodnota.

Čím více komparátorů síť obsahuje, tím je z našeho pohledu horší. Tato vlastnost ovšem nemá takovou váhu, jako správnost řazení. To musí být ve fitness funkci zohledněno, aby nedocházelo k tomu, že algoritmus bude upřednostňovat sítě, které jsou z hlediska počtu komparátorů sice velmi úsporné, zato však nedokáží seřadit správně všechny vstupy.

Pokud chceme, aby výsledná síť byla škálovatelná, nestačí pro určení kvality genu zkonstruovat pouze jednu síť. Musíme jich zkonstruovat několik pro různý počet vstupů, určit kvalitu každé ze sítí a celkovou fitness genotypu určíme jako součet jednotlivých dílčích údajů o kvalitě.

Kvalitu sítě o určitém počtu vstupů vypočteme podle vzorce:

$$f_w = -pocetKomparatoru - a \cdot c + k,$$

kde c značí počet správě seřazených posloupností, a koeficent, kterým násobíme počet správně seřazených vstupů pro zvýšení váhy tohoto parametru a k číslo, jehož hodnota je 500 v případě, že $c = 2^w$, jinak 0. Jeho účelem je zvýhodnění korektních řadicích sítí. Celková fitness je pak:

$$fitness = f_6 + f_7 + f_8 + f_9,$$

kde f_i, $i \in \{6, 7, 8, 9\}$ jsou hodnoty fitness pro jednotlivé sítě o i vstupech. Tato funkce byla stanovena experimentálně. Menší počet ověřovaných sítí by méně zvýhodňoval škálovatelné sítě. Pro dosažení maximální fitness by totiž stačilo, aby řešení konstruovalo platné řadicí sítě o šesti až osmi vstupech, nyní je vyžadována platnost i devítivstupé sítě. Naopak větší počet ověřovaných sítí by už příliš zatěžoval výpočet.

Mějme například koeficient $a = 2$, potom fitness řešení konstruující řadicí síť založenou na principu vkládání bude dáno rovnicí 1.

$$\begin{aligned} &2 \cdot 2^6 + 500 - 15 + 2 \cdot 2^7 + 500 - 21 + 2 \cdot 2^8 + 500 - 28 + 2 \cdot 2^9 + 500 - 36 = \\ &= 1920 + 2000 - 100 = 3820 \end{aligned} \tag{1}$$

5.1.3 Výběr jedinců

Pro výběr jedinců do reprodukčního procesu jsem využil *turnajový výběr*. Z populace se vyberou 4 náhodní jedinci, z nichž nejlepší dva se stanou rodiči. Tento proces se opakuje tak dlouho, dokud nemáme dostatečně velkou množinu rodičů.

5.1.4 Křížení

Pro křížení chromozomů jsem dal přednost *jednobodovému* a *dvoubodovému křížení* oproti křížení *uniformnímu*. Ty zachovávají pravidla (tzn. ponechává stejné pravidlo, které má některý z rodičů), ve kterých se nenachází bod křížení, což se v tomto případě o uniformím křížení říci nedá.

Uniformní křížení udělá z chromozomu často naprosto jiný. Změnou jediného bitu pravděpodobně získáme odlišné pravidlo. Jak jsme si uvedli v kapitole 2.3.3, uniformní křížení bity potomka náhodně skládá z bitů jednoho nebo druhého rodiče. Pravděpodobnost, že takto zkřížíme gen tak, aby reprezentoval pravidlo jednoho z rodičů je $P = (P_1)^n$, kde n je počet bitů v genu, $P_1 = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$ je pravděpodobnost, že bit obou rodičů na daném místě je stejný ($P_s = \frac{1}{2}$) nebo je různý a vybere se bit z prvního rodiče ($P_d = \frac{1}{2 \cdot 2}$). Což v případě genů délky 6 činí $P = \frac{3^6}{4^6} = \frac{729}{4096} \doteq 0.178$. Vezmeme-li v úvahu možnost tiché mutace[1] vzniklé při uniformním křížení, jejíž pravděpodobnost je pravděpodobnost, že se změní pouze a jen nejvyšší bit, čímž se hodnota genu zvýší nebo sníží o 32 (což je přesně použitý počet možných pravidel) je $P_{sm} = \frac{1}{4} \cdot (\frac{3}{4})^5 = 0.0593$, pak pravděpodobnost, změny každého genu tak, že nereprezentuje pravidlo z žádného z rodičů je $P' = \frac{1}{0.178 + 0.0593}$, tedy asi 4.213 ku jedné. To znamená, že při použití uniformního křížení se teoreticky při chromozomu délky 8 zachovají pouze 1 až 2 pravidla z původních pravidel některého z rodičů.

U křížení se nevyužívá pravděpodobnost křížení. Algoritmus obnovy populace je navržen tak, že do následující generace posouvá i část původní populace. U části, která je určena ke křížení, se žádný z rodičů při křížení nevynechá. Pravděpodobnost křížení je tedy u této části vždy 100%, proto při popisu experimentů tuto pravděpodobnost nebudeme explicitně uvádět.

5.1.5 Mutace

Míra mutace m se udává v čísle v rozmezí 0 až 1000. Mutace probíhá vygenerováním náhodného čísla pro každý gen. Je-li toto číslo nižší, než nastavená míra mutace, bit genu je negován. Pravděpodobnost tiché mutace je $P_{sm} = \frac{m}{1000} + (\frac{1000 - m}{1000})^{(n-1)}$.

5.1.6 Obnova populace

Populace je obnovována z části potomky, z části původní populací a protože je stavový prostor značně členitý, využil jsem začlenění *imigrantů* do další generace. Tedy náhodně vygenerovanými chromozomy, kterých je 10% populace. Imigranti nahrazují jedince s nejhorší fitness. Využívá se také *elitismu*, kde nejlepší jedinec postoupí do další generace. S elitismem program vykazoval kvalitnější výsledky i rychlejší konvergenci.

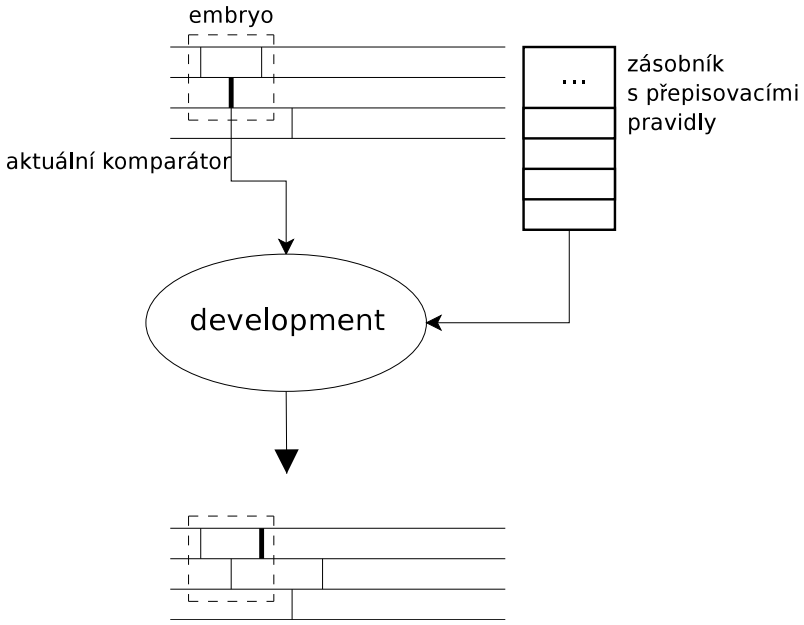
5.2 Development

V developmentu se s využitím přepisovacích pravidel obsažených v genotypu postupně ve vývojových krocích rozvíjí počáteční řešení – embryo. Algoritmus hledá pravidla tak, aby se aplikací všech pravidel v genotypu vytvořila řadicí síť, která má o určitý počet vstupů více než embryo. Počet přidaných vstupů budeme označovat jako *velikost kroku developmentu* s_d.

[1] V tomto případě se však jedná o změnu hodnoty genu při křížení, nikoliv mutaci.

V procesu vývoje embrya hrají roli tři struktury – genotyp, aktuálně zpracovávaný komparátor a samotná síť komparátorů. Za aktuálně zpracovávaný komparátor je na počátku vývoje prohlášen nejlevější komparátor sítě, v průběhu může být pravidly nastavován na další komparátory. Síť komparátorů je na začátku vývoje shodná s embryem pouze s tím rozdílem, že má o daný počet vstupů více. Cílem developmentu je z této sítě (která díky přidaným vstupům nedokáže seřadit vstupní vektor do neklesající posloupnosti) sestrojit platnou řadicí síť.

Na obr. 11 vidíme graficky znázorněn vývojový krok. Máme síť komparátorů a tučnou čarou označen aktuálně zpracovávaný komparátor. Pravidla jsou uložena v zásobníku, provádí se vždy pravidlo na vrcholu. Po provedení se ze zásobníku odstraní. Interpretací pravidla se transformuje síť, případně i změní aktuálně zpracovávaný komparátor (záleží na pravidle).



Obrázek 11: Znázornění developmentu.

Na aplikaci pravidla se dá pohlížet jako na zobrazení $r : N \times \mathbb{N} \to N \times \mathbb{N}$, kde prvek z množiny N představuje síť komparátorů a prvek z množiny přirozených čísel $\mathbb{N}$ index aktuálně zpracovávaného komparátoru. Vývojový krok je kompozice $d = r_n \circ r_{n-1} \circ ... \circ r_1$, kde n značí celkový počet pravidel. Celý development je tedy funkce složená kompozicí vývojových kroků. Z toho je zřejmé, že konstrukce řadicí sítě s počtem vstupů, který přesahuje počet vstupů embrya o více než jeden, tedy development o více krocích, není totéž jako provedení jednoho vývojového kroku a použití výsledné sítě jako embrya pro provedení dalšího vývojového kroku. Rozdíl je ve způsobu zacházení s aktuálním komparátorem. V prvím případě se proces vývoje sítě chová jako jedna funkce $N \times \mathbb{N} \times R^* \to N \times \mathbb{N}$ (R značí množinu pravidel), aktuální komparátor se tedy nastaví pouze před jejím provedením a jeho další pohyb záleží pouze na použitých pravidlech. Kdežto v druhém případě se nastavuje před každým vývojovým krokem opět na nejlevější komparátor.

Během developmentu se snadno může stát, že se v síti objeví komparátory, které nemají na funkci sítě vliv. Mohou však pomoci při tvorbě sítě nebo samy být upraveny do podoby, která již vliv má. Po dokončení vývoje jsou již zbytečné a ze sítě se odstraní.

Komparátory, které se po developmentu ze sítě odeberou jsou všechny, které splňují alespoň jednu z těchto podmínek:

- Komparátor je identický se sousedním,
- má oba vstupy připojeny na stejný vstup sítě,
- je připojen do sítě pouze jedním vstupem – jeden z jeho vstupů „přesahuje" mimo síť.

Je zřejmé, že jde o optimalizaci, která je specifická pro řešenou úlohu a není součástí evolučního algoritmu. Přesto je tento krok součástí popisovaného algoritmu pro návrh řadicích sítí, proto je třeba brát v úvahu, že později prezentované výsledky jsou sítě po odebrání nepotřebných komparátorů.

5.2.1 Typy pravidel pro rozvoj sítě

Využijeme-li analogie k biologickému developmentu, můžeme pravidla rozdělit do skupin na pravidla zajišťující:

- *Růst* – podíváme-li se na zjednodušené grafické vyjádření řadicí sítě, vidíme komparátory jako úsečky o určité délce. Pokud změníme některý ze vstupů nebo výstupů, úsečka změní svou velikost. Z tohoto pohledu můžeme pravidly nechat komparátor „růst", tedy připojovat ho na jiné vstupy a tím tak ovlivňovat funkci sítě.
- *Dělení* – tato skupina pravidel zvyšuje celkovou velikost sítě. Jejich funkce je stejná jako biologickém developmentu. Dělení může být symetrické i asymetrické. U symetrického dělení se komparátory dělí na dva stejně velké. U řadicích sítí jde spíše o klonování, kdy se komparátor zkopíruje v nezměněné podobě. Asymetrické dělení naopak způsobí, že nově vzniklý komparátor je větší nebo menší (ve smyslu popsaném v přechozím bodu), než původní.
- *Diferenciaci* – jak jsme si popsali v kapitole 4.1, diferenciace je proces, díky kterému buňky získávají tvar podle konkrétního účelu a tvoří tak jednotlivé orgány. Zde je diferenciace reprezentována přebíráním podoby okolních komparátorů.
- *Formování tvaru* – díky těmto pravdlům se komparátory v rámci sítě přesouvají. Mohou se přesouvat *horizontálně* – např. se sousedním komparátorem vyměnit pozici, přesunout nakonec, nebo *vertikálně* – přepojení komparátoru na jiné vstupy sítě. Přesunutí na konec sítě se v případě aplikace v řadicích sítích, často využívá např. s dělením – nový komparátor se nevytvoří vedle původního, ale na poslední pozici v síti. Toto se také často kombinuje s vertikálními posuvy.
- *Řízení* – stejně jako proteiny v biologickém developmentu nemají vliv pouze na buňky, ale i na další proteiny nebo i samotný genotyp, tak i zde exisutuje skupina takových pravidel. Typickým představitelem je pravidlo, které zmultiplikuje následující pravidlo v genotypu např. v závislosti na počtu vstupů sítě. Mohou se vyskytovat pravidla, která např. v závislosti na sudosti počtu vstupů sítě potlačí následující pravidlo apod.

Pravidla se dají také rozdělit podle jiného kritéria. Existují *elementární* pravidla a pravidla *složená*, což je posloupnost pravidel elementárních. Většina pravidel je složených, abychom

zmenšili velikost prohledávaného prostoru. Proto si pravidlo zobecněme jako *seznam elementárních pravidel*. Jako pravidlo bude dále v textu uvažujme tento seznam elementárních pravidel, byť i o jednom prvku.

Protože v tomto modelu neexistuje nic jako shlukování pravidel, je zařazení složených pravidel výhodné v kombinaci s řídícími pravidly – takové pravidlo ovlivňuje pouze jedno následující pravidlo, takže v případě, že by místo složeného pravidla byla posloupnost elementárních pravidel se stejnou funkcí, pak by byla celková funkce odlišná. Tento přístup je také podobnější biologickému developmentu v tom, že se jednotlivé děje provádějí současně.

5.2.2 Použitá pravidla

Podívejme se nejdříve na elementární pravidla a definujme jejich funkci.

- *next*. Toto pravidlo posune aktuálně zpracovávaný komparátor na jeho pravého souseda. Pokud však aktuálně zpracovávaný komparátor je poslední v síti, toto pravidlo nemá žádný efekt.
- *clone*. Zkopíruje aktuální komparátor a nově vytvořený komparátor vloží bezprostředně za něj.
- *appendClone*. Klonuje aktuálně zpracovávaný komparátor a nově vzniklý zařadí na konec sítě.
- *moveDown/moveDownClone*. Posune komparátor v grafické reprezentaci směrem dolu, tedy připojí oba vstupy i výstupy komparátoru na vstupy a výstupy sítě s indexem o jedna vyšším, než na které jsou aktuálně připojeny. Pokud by se touto operací měl komparátor dostat za poslední vstup sítě, je přesunut tak, že jeho první vstup je připojen na první vstup sítě. Pravidlo *moveDown* je platné pro aktuální komparátor, *moveCloneDown* pro nově naklonovaný komparátor vznikne po aplikaci pravidla *clone* nebo *appendClone*. Sémantika ostatních pravidel s příponou -*Clone* je stejná jako v tomo případě.
- *moveDown'/moveDownClone'*. modifikace předchozích pravidel. Funkce se liší v ošetření situace, kdy je vstup komparátoru posunut mimo síť. Tato pravidla vstupy komparátorů posunou na následující vstupy sítě a pak je operací modulo přizpůsobí.
- *increase/increaseClone*. Zvýší „velikost" komparátoru, ve smyslu popisovaném v odstavci 5.2.1. Přiřadí tedy druhý vstup a výstup komparátoru na následující index vstupního vektoru. Pokud je komparátor již přiřazen na poslední vstup, nemá toto pravidlo žádný vliv.
- *increase'/increaseClone'*. Modifikace pravidla *increase*. Funkce se liší v ošetření situace, kdy je komparátor připojen na poslední vstup. Toto pravidlo připojí daný vstup komparátoru na vstup sítě daný operací: index požadovaného vstupu sítě *modulo* počet vstupů sítě.
- *decrease/decreaseClone*. Sníží „velikost" komparátoru. Pokud má komparátor již „velikost" 1 (tzn. je připojen na sousední dva vstupy sítě), nemá toto pravidlo vliv.
- *split*. Toto pravidlo rozdělí komparátor na 2 stejně velké komparátory. Provede se pouze v případě, že je to možné, tedy komparátor musí mít sudou „velikost". Poté

komparátor zmenší na poloviční velikost a duplikuje. Duplikát se vloží před aktuální komparátor.

Tabulka 1 zobrazuje všechna pravidla, které se využívala v evolučním procesu. Pro snížení členitosti prohledávaného prostoru řešení jsem se snažil, aby pravidla s podobnou funkcí měla co nejmenší *hammingovu vzdálenost*, tedy aby se jejich binární reprezentace lišila o co nejmenší počet bytů. Proto jsou sousední pravidla seřazeny podle jejich „podobnosti" a jejich číselná reprezentace odpovídá příslušnému binárnímu řetězci v posloupnosti dané *Grayovým kódem*.

Modifikace sítě aplikací složeného pravidla je atomická – provádí se transakčně. Pokud některé z elementárních pravidel nelze provést, neprovede se žádné z elementárních pravidel modifikující síť.

Můžeme si všimnout, že zavádíme i bloky posloupností pravidel, které byly v evoluci úspěšné. Jde o pravidla 22, 6 a 2. Z pohledu jejich sémantiky to jsou *makra*. Při jeho použití se přepíše pravidly, které obsahuje. Z toho plyne rozdíl oproti běžným složeným pravidlům. Pravidla obsažená v těchto makrech se provádí nezávisle. Pokud tedy selže jedno, ostatní se vykonat mohou.

Zavedením maker zvýšíme pravděpodobnost, že se jednotlivá pravidla vyskytnou v tomto pořadí. Výhodou je upřednostnění kvalitních posloupností pravidel. To se však dá považovat i za nevýhodu, protože tím ubíráme na obecnosti řešení. Aby záporný vliv maker nebyl příliš znatelný, jsou makra složená pouze ze dvou pravidel.

Pravidel je celkem 32, pravidla 0 a 31, která nejsou v tabulce uvedena, nemají žádnou funkci. Genotyp má pevně danou délku, ale s největší pravděpodobností existují řešení, která nebudou potřebovat tolik pravidel, kolik daný genotyp umožňuje. Proto zavádíme pravidla bez funkce, která vyplní tyto přebytečné geny.

Pravidla nejsou ve tvaru $A \rightarrow B$, jak je zvykem, protože se pro danou doménu neukázala příliš vhodná. Těmito pravidly se nepodařilo nalézt způsob konstrukce škálovatelných řadicích sítí. Dále se zavedením řídicích pravidel, jako je replikace následujícího pravidla v závislosti na aktuálním počtu vstupů sítě, kvalita řešení podstatně zlepšila. Tím se také zvýšila podobnost pravidel s popsanou funkcí proteinů v developmentu i když na úkor možnosti popsat přepisovací pravidla některou z běžných gramatik. Stále však jde o přepis struktury reprezentující síť aplikací pravidla na novou strukturu, jde tedy stále o přepisovací systém.

#	binárně	funkce
16	10000	appendClone, moveDownClone, moveDownClone, increaseClone, next
20	10100	appendClone, moveDownClone, moveDownClone, decreaseClone, next
28	11100	appendClone, moveDownClone, moveDownClone, increaseClone', next
30	11110	appendClone, moveCloneDown, moveCloneDown, decreaseClone, decreaseClone, next
26	11010	appendClone, moveCloneDown, moveCloneDown, decreaseClone, next
24	11000	appendClone, next
8	01000	clone, moveDownClone', next
10	01010	appendClone, moveCloneDown
14	01110	appendClone, moveCloneDown, moveCloneDown
15	01111	appendClone, moveCloneDown, next
13	01101	appendClone, moveCloneDown, moveCloneDown, moveCloneDown, increaseClone
12	01100	appendClone, moveDownClone, moveDownClone, moveDownClone, decreaseClone
4	00100	clone, increaseClone',next,next
5	00101	split, next
7	00111	decrease
23	10111	increase
22	10110	29, 15
6	00110	15, 11
2	00010	14, 21
18	10010	next
19	10011	(w-3) × [appendClone, next]
27	11011	(w-2) × [appendClone, next]
11	01011	(w-1) × [appendClone, next]
3	00011	(w-4) × [appendClone, next]
1	00001	appendClone, moveDown, next
9	01001	replicate next rule (w-2)×
25	11001	replicate next rule (w-3)×
29	11101	appendClone, increaseClone'
21	10101	appendClone, moveCloneDown, increaseClone'
17	10001	appendClone, moveCloneDown, moveCloneDown, decreaseClone, decreaseClone, next

Tabulka 2. Seznam pravidel zůčastněných v evolučním procesu.

Kapitola 6

Experimenty

Program provádějící experimenty byl napsán v jazyce Haskell s využitím kompilátoru ghc 6.6 a prostředí operačního systému FreeBSD. Vše, snad až na využitou standardní grafickou knihovnu HGL[1], je přenositelné a o testováno v OS Linux. Nevyžaduje žádné nestandardní externí knihovny, genetický algoritmus i development jsou součástí programu. Použití programu včetně příkladů ze školního prostředí naleznete v přiloženém uživatelském manuálu. Program není zamýšlen jako uživatelská aplikace ani produkční systém, jde spíše o prototyp, není tedy optimalizován pro maximální výkon.

6.1 Pevně daná embrya

Za pomoci evolučního algoritmu s přímým zakódováním bylo vyevolvováno několik embryí o dvou, třech, čtyřech a také jedno embryo o pěti vstupech. Tato embrya pak byla využívána v experimentech, jejichž cílem bylo vybrat několik nejvhodnějších pravidel pro konstrukci škálovatelných řadicích sítí. Protože není výpočetně možné ověřit, zda daná posloupnost pravidel generuje síť s neomezeně velkým počtem vstupů, provede se 10 vývojových cyklů a pokud jsou všechny vygenerované sítě řadicí, je řešení prohlášeno za škálovatelné.

6.1.1 Experiment I

První experimenty směřovaly na návrh sítě s libovolným počtem vstupů. Krok developmentu s_d byl tedy 1, každý vývojový krok měl o jeden vstup více než embryo.

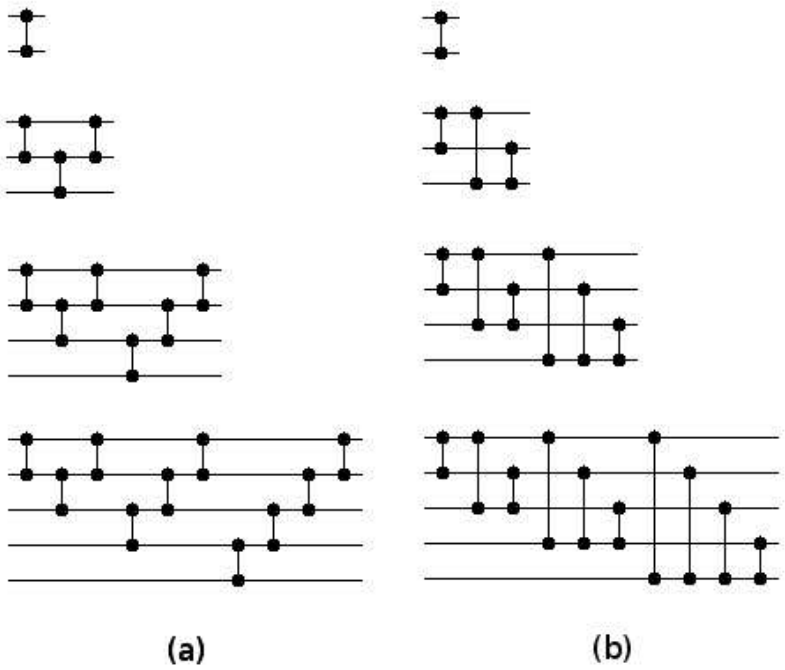
Pro každé z embryí bylo provedeno 100 evolučních cyklů o 200 generacích s populací o počtu 100 jedinců, pravděpodobnost mutace 0.08% za použití jednobodového křížení. Délka chromozomu byla nastavena na 8 pravidel.

Souhrn experimentů vidíme v tabulce 2. Kromě embrya tabulka obsahuje pro informaci počet komparátorů pro různé počty vstupů, ze kterých se skládá nejlepší nalezená síť. Dále obsahuje informaci o *úspěšnosti*. Tou je míněno procento případů, ve kterých se podařilo najít sadu přepisovacích pravidel, které z daného embrya dokáží zkonstruovat škálovatelnou síť.

Nejlepší navržená síť pro libovolný počet vstupů jak z hlediska zpoždění tak počtu komparátorů, byla shodná se sítí zkonstruovanou na principu vkládání/výběru. Byly však nalezeny dva různé principy. Nejdříve bylo znovuobjeven princip vkládání, poté trochu odlišný

[1] HGL není zcela přenostielná na MS Windows.

způsob řazení, který má stejné parametry jak z hlediska počtu komparátorů tak zpoždění. Několik jejich vývojových kroků vidíme na obrázcích 12a a 12b.

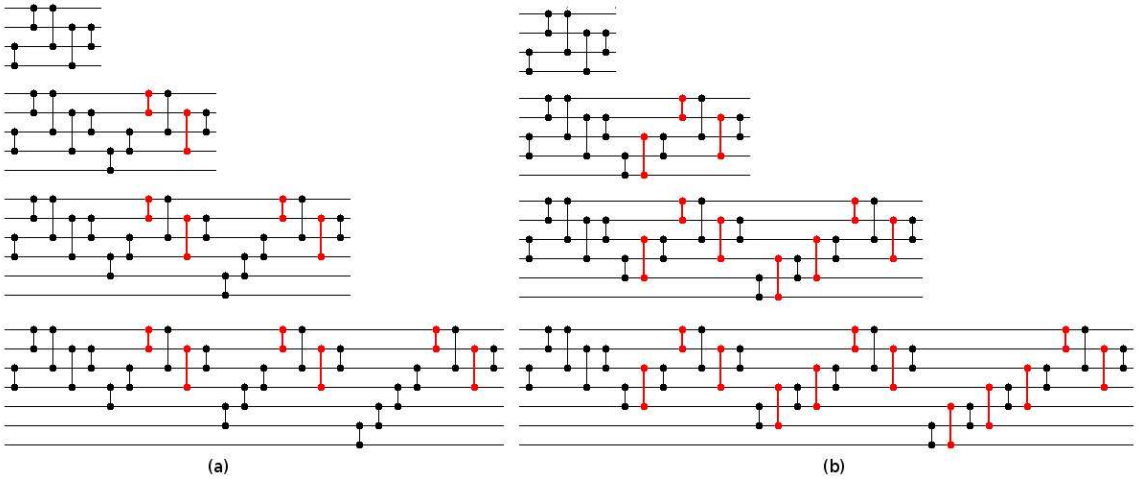


Obr. 12: Příklad konstrukce řadicí sítě: (a) znovuobjevený princip řazení sestrojený např. posloupností pravidel (10, 17, 27, 6, 11, 18, 17), (b) nový stejně dobrý princip sestrojený pravidly (22, 25, 15, 13, 6). Dvouvstupá síť nahoře znázorňuje embryo.

V některých nalezených řešeních však nejsou některé komparátory využity, tzn. při seřazení všech možných vstupů se nepoužije k výměně hodnot na vstupech ani jednou. Jsou tedy *redundantní*. S vypuštěním těchto komparátorů se některá řešení vyznačují menším počtem komparátorů, než jaký vyžaduje konvenční řešení. Počet nutných komparátorů (tzn. po vypuštění redundantních) je v tabulce udán číslem v závorce. Příklady konstrukce sítí, které po odstranění redundantních komparátorů mají lepší parametry než konvenční síť, vidíme na obrázku 13a a 13b.

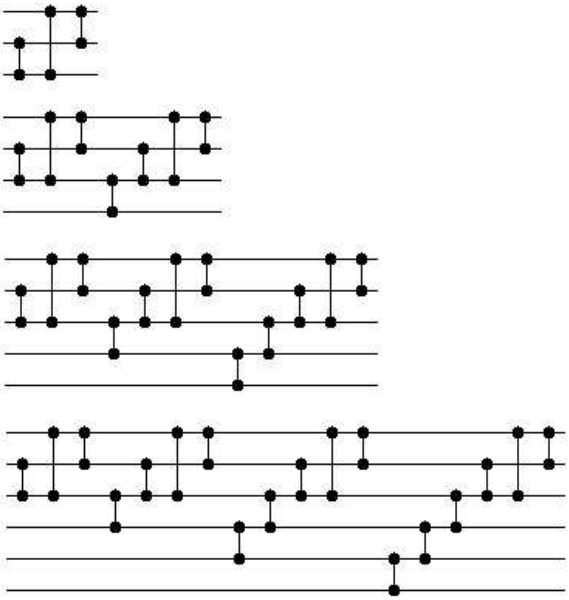
Tabulka 2: Souhrn experimentů s pevně daným embryem a $s_d = 1$. Údaje v závorce udávají počet nutných komparátorů (bez redundantních).

Embryo	**Pravidla**	počet kompátorů pro daný počet vstupů							**Úspěšnost**
		6	**7**	**8**	**9**	**10**	**11**	**12**	
(0,1)	10,17,27,6,11,18,17	15 (15)	21 (21)	28 (29)	36 (36)	45 (45)	55 (55)	66 (66)	100%
(1,1)(0,2)(0,1)	18,28,14,10,19,30,27,22	15 (15)	21 (21)	28 (28)	36 (36)	45 (45)	55 (55)	66 (66)	100%
(0,1)(1,1)(0,1)	24,13,10,27,7,7,17,28	15 (15)	21 (21)	28 (28)	36 (36)	45 (45)	55 (55)	66 (66)	100%
(0,1)(0,2)(1,1)	15,31,29,9,15,20,25,14	15 (15)	21 (21)	28 (28)	36 (36)	45 (45)	55 (55)	66 (66)	75%
(2,1)(0,1)(0,2)(1,2)(1,1)	15,10,5,0,29,15,25,15	18 (14)	26 (20)	35 (27)	45 (35)	56 (44)	68 (54)	81 (65)	82%
(0,3)(1,1)(2,1)(0,1)(1,1)	13,29,17,18,28,10,14,27	16 (14)	23 (20)	31 (27)	41 (35)	52 (44)	64 (54)	77 (65)	53%
(0,3)(1,3)(0,2)(1,2)(2,2)(0,1) (3,1)(1,1)(2,1)	28,0,29,11,10,19,26,9	20 (14)	33 (20)	47 (27)	64 (35)	82 (44)	103 (54)	126 (65)	6%



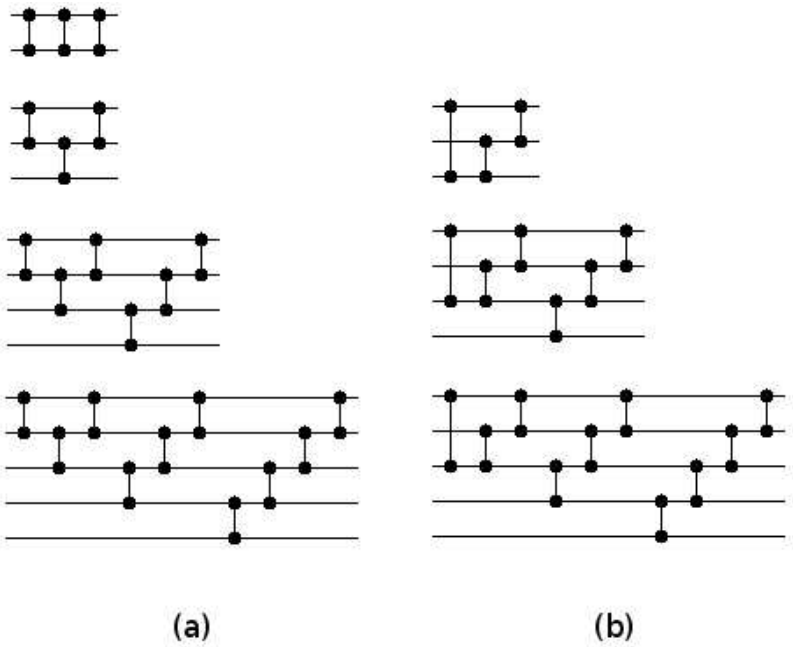
Obr. 13: Konstrukce řadicích sítí, které jsou po odebrání redundantních komparátorů efektivnější než konvenční řešení. Červeně označené komparátoy jsou nadbytečné. Čtyřvstupá síť nahoře znázorňuje embryo. Sítě jsou sestrojené pomocí pravidel: (a) (10,14,25,14,10,27,0,19), (b) (10,29,27,3,24,3,8,28).

Z hlediska evoluce se princip vkládání ukázal velmi výhodný. Podobný princip se uplatňuje i za použití jiných embryí, jak vidíme např. na obrázku 14.



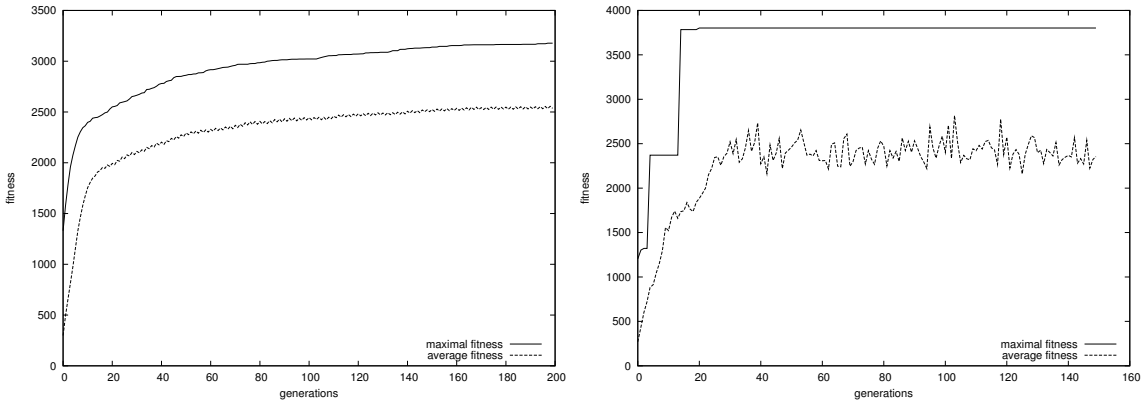
Obr. 14: Princip podobný vkládání s využitím jiného embrya. Síť byla zkonstruována pravidly (10, 11, 16, 31, 13, 9, 16).

Dále bylo provedeno několik dalších pokusů s jinými embryi. Zajímavostí je například genotyp, který si dokázal upravit embryo do podoby vhodné pro tento způsob řazení, jak vidíme na obr. 15a, nebo genotyp, který použil embryo s jinak uspořádanými komparátory, ale přesto našel cestu, jak z něj v nezměněné podobě vyrobit zbytek sítě naprosto shodný se sítí zkonstruovanou na základě principu vkládání (obr. 15b).



Obr. 15: Příklad konstrukce řadicí sítě sestrojené pravidly: (a) (13,5,10,10,11,17,1), (b) (28, 10, 27, 16, 23, 11, 7, 18).

Na obrázku 16a máme znázorněnu závislost průměrné a maximální fitness na počtu proběhlých generací. Hodnoty zanesené do grafu vznikly zprůměrováním hodnot ze všech experimentů. Vidíme, že se hodnota fitness nejlepšího jedince v prvních deseti až dvaceti generacích velmi rychle zvyšuje, což reflektuje nalezení řešení, které je schopno zkonstruovat v několika málo prvních vývojových krocích síť, která je schopna seřadit správně všechny vstupní hodnoty. V dalších vývojových krocích však zkonstruovaná síť není platnou řadicí sítí, řešení tedy není škálovatelné. V průběhu přibližně dalších 100 generací se fitness pomalu zvyšuje až začne stagnovat. Většina kandidátních řešení s fitness větší než 3000 je škálovatelná.



(a) (b)

Obrázek 16: Závislost fitness na počtu generací u experimentů s pevně daným embryem a $s_d = 1$: (a) průměrný průběh, (b) příklad průběhu fitness pro jeden běh algoritmu s embryem (0,1).

Řekli jsme si, že fitness řešení generujícího síť podle principu vkládání je 3820, která se v grafu nevyskytuje. Nutno si však uvědomit, že graf obsahuje průměr všech experimentů s pevně daným embryem, počítají se do něj tedy i experimenty s nízkou úspěšností, kde fitness přesáhla hodnotu 3000 jen zřídka.

Na obr. 16b vidíme tento průběh pro jeden běh algoritmu. V tomto případě je využíváno embryo (0,1).

6.1.2 Experiment II

Protože většina výsledků prvního experimentu byla velmi blízká řadicí síti sestrojené konvenčním způsobem, bylo potřeba provést další sadu experimentů. Abychom zlepšili parametry sítě, zkusíme zvýšit velikost kroku s_d na 2 vstupy.

Kvůli vyššímu kroku developmentu mohou nalezená pravidla konstruovat pouze sítě s lichým nebo sudým počtem vstupů v závislosti na velikosti embrya. Proto se dříve popsaná fitness funkce nedá aplikovat. Nepoužijeme proto součet fitness sítí o pevně daném počtu vstupů, ale sestrojíme čtyři vývojové stupně sítě po embryu. Fitness f' za použití embrya o počtu vstupů w_e a krokem developmentu s bude mít tvar:

$$f' = f_{w_e+s} + f_{w_e+2s} + f_{w_e+3s} + f_{w_w+4s}$$

Opět bylo provedeno 100 evolučních procesů pro několik různých embryí. Experimentálně se ukázalo, že bude potřeba vyšší počet generací. Zvoleno bylo 500 generací, velikost populace 100 jedinců, pravděpodobnost mutace 0,1% a dvoubodové křížení. Nejlepší výsledky vidíme v tabulce 3 pro sudý počet vstupů a v tabulce 4 pro lichý počet vstupů.

Experimenty s krokem developmentu 2 vykazovaly výrazně nižší úspěšnost. Povedlo se však nalézt několik dobrých řešení. Zejména síť na obrázku 18a, která vyžaduje nejmenší počet komparátorů ze všech provedených experimentů a je efektivní i z hlediska zpoždění. Také sítě na obrázích 17 a 18b se vyznačují nižším počtem komparátorů, než je potřeba u řešení konstruovaném konvenční metodou založenou na principu vkládání. Po odstranění redundandních komparátorů má síť z obrázku 18b stejný počet komparátorů, jako síť 18a. Konkrétní zlepšení oproti konvenčnímu řešení v počtech komparátorů zjistíme porovnáním tabulek s nejlepšími výsledky jednotlivých experimentů (tab. 2, 3 a 4).

Tabulka 5 ukazuje parametry dalších škálovatelných sítí při různých počtech vstupů zkonstrované nalezenými posloupnostmi pravidel.

Tabulka 3: Nejlepší výsledky experimentů s pevně daným embryem a $s_d = 2$ pro embrya se sudým počtem vstupů.

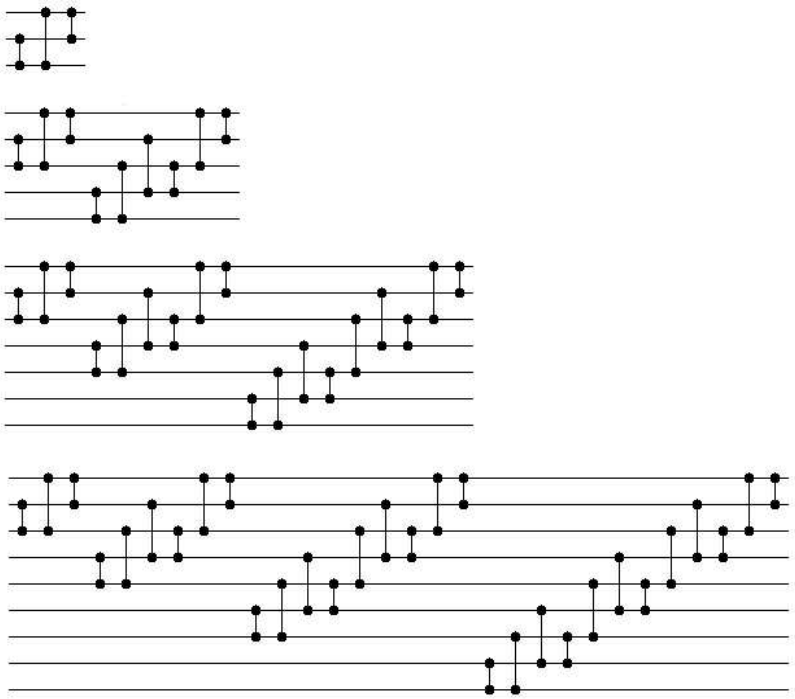
		poč. komparátorů na poč. vstupů					
Embryo	**Pravidla**	**6**	**8**	**10**	**12**	**14**	**Úspěšnost**
(0,1)	konvenčně sestroj. síť [15]	15	28	45	66	91	-
(0,1)	2, 22, 19, 24, 25, 24, 29, 1	12 (12)	22 (22)	35 (35)	51 (51)	70 (70)	11%
(2,1)(0,1)(0,2)(1,2)(1,1)	2, 22, 27, 25, 24, 21, 30, 28	13 (12)	24 (22)	38 (35)	55 (51)	75 (70)	6%

Tabulka 4: Nejlepší výsledky experimentů s pevně daným embryem a $s_d = 2$ pro embrya s lichým počtem vstupů.

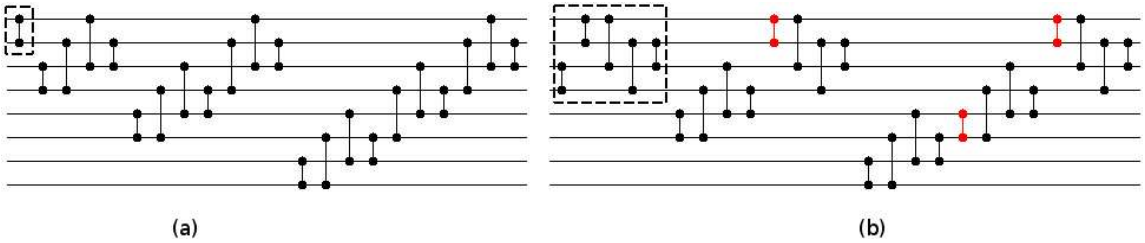
		počet komparátorů na poč. vstupů					
Embryo	**Pravidla**	**5**	**7**	**9**	**11**	**13**	**Úspěšnost**
(0,1)	konvenčně sestroj. síť [15]	10	21	36	55	78	-
(1,1)(0,2)(0,1)	2, 22, 19, 9, 24, 21, 20	9 (9)	18 (18)	30 (30)	45 (45)	63 (63)	9%

Tabulka 5: Výběr výsledků experimentů s pevně daným embryem a $s_d = 2$.

		počet komparátorů na poč. vstupů								
Embryo	**Pravidla**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
(0,1)	konvenč. sestrojená síť [15]	15 (15)	21 (21)	28 (29)	36 (36)	45 (45)	55 (55)	66 (66)	78 (78)	91 (91)
(1,1)(0,2)(0,1)	2, 22, 19, 9, 24, 21, 20	-	18 (18)	-	30 (30)	-	45 (45)	-	63 (63)	-
(1,1)(0,2)(0,1)	2,29,15,31,31,21,9,3	-	21 (18)	-	36 (30)	-	55 (45)	-	78 (63)	-
(1,1)(0,2)(0,1)	9,14,2,22,9,19,19	-	19 (18)	-	34 (30)	-	54 (54)		79 (63)	-
(1,1)(0,2)(0,1)	2,10,29,3,0,24,3,15	-	21 (21)	-	36 (36)	-	55 (55)	-	78 (78)	-
(1,1)(0,2)(0,1)	9,14,2,22,9,11,20,13	-	22 (18)	-	38 (30)	-	59 (45)	-	85 (63)	-
(0,1)	12,2,29,22,12,9,3,19,20	12 (12)	-	22 (22)	-	35 (35)	-	51 (51)	-	75 (70)
(0,1)	14,2,22,17,9,3	13 (12)	-	25 (22)	-	41 (35)	-	61 (51)	-	85 (70)
(2,1)(0,1)(0,2)(1,2)(1,1)	2,22,27,25,24,21,30,28	13 (12)	-	24 (22)	-	38 (35)	-	55 (51)	-	75 (70)
(2,1)(0,1)(0,2)(1,2)(1,1)	14,2,22,17,9,3	13 (12)	-	25 (22)	-	41 (35)	-	61 (51)	-	85 (70)
(2,1)(0,1)(0,2)(1,2)(1,1)	2,22,21,27,9,24,22,21	14 (13)	-	27 (24)	-	44 (38)	-	65 (55)	-	90 (75)
(2,1)(0,1)(0,2)(1,2)(1,1)	9,14,2,22,9,11,28	14 (12)	-	28 (22)	-	47 (35)	-	71 (51)	-	100 (70)

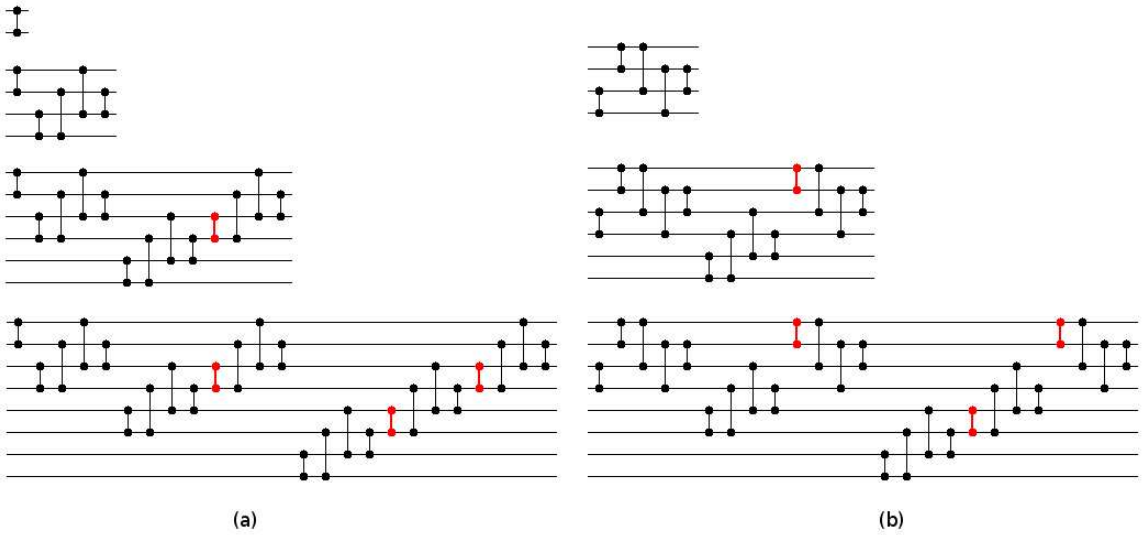


Obrázek 17: Konstrukce efektivní řadicí sítě za pomocí pravidel (2, 22, 19, 9, 24, 21, 20).



Obrázek 18: Příklady efektivních řadicích sítí vyevolvovaných s $s_d = 2$ za pomocí pravidel (a) (2, 22, 19, 24, 25, 24, 29, 1) (b) (14, 2, 22, 17, 9, 3). Embryo je vyznačeno přerušovaným obdélníkem. Redundantní komprátory jsou označeny červeně, po jejich odstranění mají sítě shodný počet komparátorů.

Posloupnost pravidel z obrázku 18b je také zajímavá tím, že konstruuje platnou řadicí síť jak z embrya, které je na obrázku znázorněno, tak z embrya (0,1). Obě tyto sítě mají po odstranění redundantních komparátorů stejný počet komparátorů. Postup konstrukce obou sítí vidíme na obrázcích 19a a 19b.



Obrázek 19: Konstrukce sítě s $s_d = 2$ za pomoci posloupností pravidel (14, 2, 22, 17, 9, 3) s využitím embrya: (a) (0,1), (b) (2,1)(0,1)(0,2)(1,2)(1,1).

6.2 Evolvovaná embrya

Dále jsem provedl několik experimentů, kdy genotyp obsahoval kromě přepisovacích pravidel také samotné embryo. V evolučním procesu se tedy hledala vhodná kombinace embrya a pravidel, nikoli pouze vhodná pravidla pro dané embryo.

Tento přístup může navrhnout embryo jako síť komparátorů, která sama o sobě ještě není platnou řadicí sítí. Tou se stane až po vývinu embrya aplikací příslušných pravidel. Embryo by v tomto případě bylo pouze jakýsi polotovar. Je však pravděpodobné, že úspěšná embrya budou stejná jako v předchozím případě, kdy byla pevně dána a sama o sobě byla řadicí sítí. V takovém případě se experimentováním s embryem obsaženým přímo v genotypu přesvědčíme, které embryo je s danou množinou dostupných pravidel pro konstrukci škálovatelných řadicích sítí nejvhodnější.

6.2.1 Zakódování kandidátního řešení

Chromozom je opět binární řetězec, má však dvě pomyslné části. Na levé straně je zakódováno embryo, na pravé posloupnost pravidel. Způsob kódování pravidel zůstal nezměněn. Embryo je v binárním řetězci reprezentováno posloupností zakódovaných vstupů komparátorů. Komparátor je určen dvěma po sobě jdoucími vstupy.

Každý vstup komparátoru je reprezentován číslem $0 \leq x \leq 8$, které určuje, na jaký vstup sítě je připojen. Nejde přímo o index vstupu embrya, protože maximální počet jeho vstupů w_e může být menší a je předem dán. Daný vstup komparátoru je připojen na vstup s indexem $i = x \, mod \, w_e$. Hodnota w_e neznamená, že chromozom obsahuje embryo o w_e vstupech, jde pouze o maximální počet vstupů, které může mít. Ve skutečnosti může být embryo méněvstupové. Pro proces konstrukce se bere v úvahu skutečný počet vstupů embrya, nikoli hodnota w_e.

Komparátor *(a,b)* se vytvoří ze dvou vstupů i_1 a i_2 následovně:

$$a = min(i_1, i_2) \ , \ \ b = max(i_1, i_2) - a$$

Každý vstup x je zakódován binárně, zabírá tedy 3 bity. Maximální počet komparátorů v embryu je pevně dán, před použitím se ovšem optimalizuje (viz. 5.2), nemusí tedy všechny možné komparátory využít. V případě, že oba vstupy komparátoru jsou stejné, nebo je komparátor shodný s předešlým, nebere se v úvahu.

000	001	101	100	011	111	001000	000010	000110

Obrázek 20: Příklad chromozomu obsahující embryo (levá část) i posloupnost pravidel (pravá část).

Na obrázku 20 vidíme příklad chromozomu, který mimo posloupnosti pravidel obsahuje i embryo o maximálním počtu komparátorů 3 a maximálním počtu vstupů $w_e = 3$. Zde je využito všech možných komparátorů i vstupů. První komparátor se dá přečíst přímo $(0, 1)$. Druhý se vypočítá z hodnot $i_1 = 5 \, mod \, w_e = 2$, $i_2 = 4 \, mod \, w_e = 1$, bude mít tedy tvar $(1, 1)$. Po výpočtu posledního komparátoru získáme celkovou podobu embrya $(0, 1)(1, 1)(0, 1)$. Pravidla obsažená v chromozomu jsou $(8, 2, 6)$.

6.2.2 Experimenty

Přidáním embrya do genotypu jsme sice zvětšili prohledávaný prostor, ale umožnili evoluci nejen změnu pravidel tak, aby se co nejlépe hodila k danému embryu, ale také změnu embrya pro dosažení lepšího výsledku s danými pravidly. Cílem hledání byly opět posloupnosti pravidel pro generování sítě s krokem developmentu $s_d \in \{1, 2\}$. V tabulce 6 vidíme souhrn výsledků. Tabulka obsahuje stejné sloupce jako v předchozích experimentech, přibyla však informace o embryu, které se v úspěšných řešení objevovalo nejčastěji. Tabulka 7 opět ukazuje některá nalezená řešení s počty komparátorů výsledných sítí včetně počtu bez redundantních komparátorů. Všechna řešení v tabulce 7 vyznačují stejným nebo nižším počtem potřebných komparátorů (tedy nepočítáme redundantní) než konvenční řešení.

Tabulka 6: Souhrn nejlepších výsledků dosažených experimenty s embryem obsaženým v genotypu.

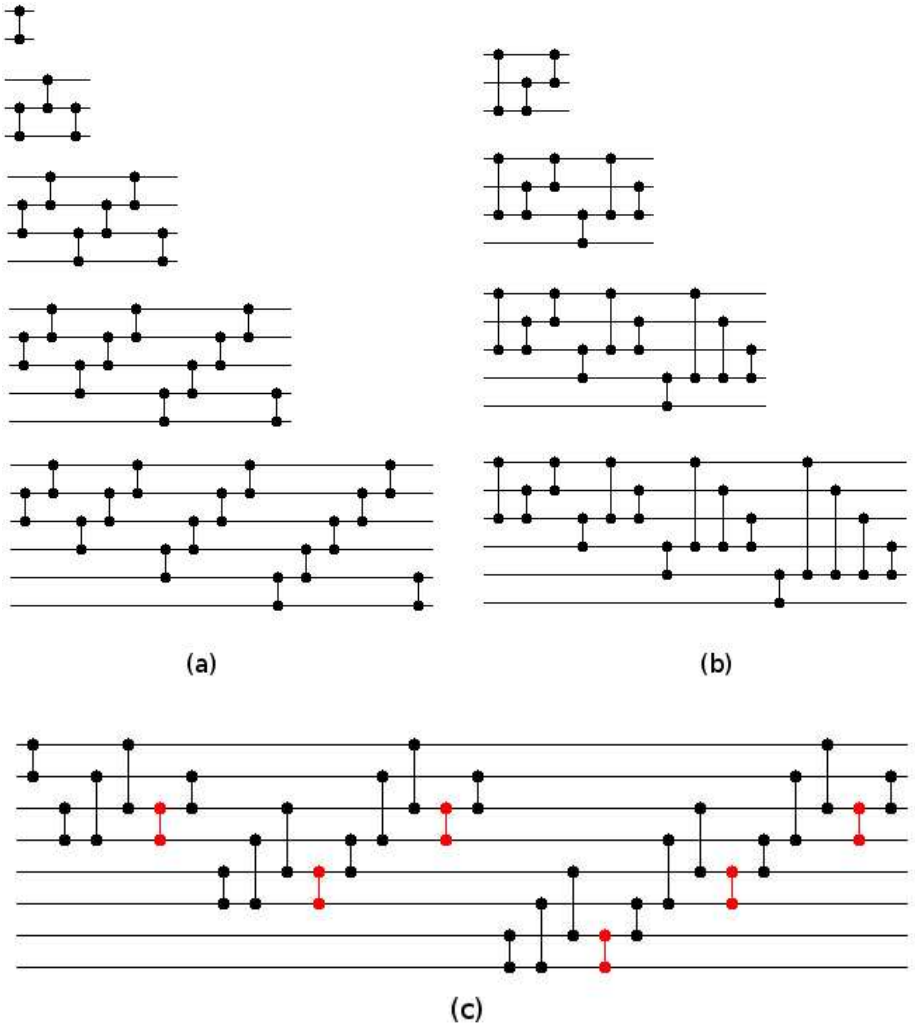
s_d	**Embryo**	**Pravidla**	počet komparátorů na poč. vstupů							**Úspěšnost**
			6	**7**	**8**	**9**	**10**	**11**	**12**	
1	(0,1)	konvenč. sestrojená síť [15]	15 (15)	21 (21)	28 (29)	36 (36)	45 (45)	55 (55)	66 (66)	-
1	(0,1)	19,10,2,8,18,27,18,14	15 (15)	21 (21)	28 (28)	36 (36)	45 (45)	55 (55)	66 (66)	100%
2	(0,1)	2,25,13,29,22,9,25,9,3,23	12 (12)	-	22 (22)	-	35 (35)	-	51 (51)	4%

Tabulka 7: Výběr výsledků dosažených experimenty s embryem obsaženým v genotypu.

s_d	**Embryo**	**Pravidla**	počet komparátorů na poč. vstupů								
			6	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
1	(0,1)	konvenč. sestrojená síť [15]	15 (15)	21 (21)	28 (29)	36 (36)	45 (45)	55 (55)	66 (66)	78 (78)	91 (91)
1	(0,1)	22,14,9,15,30,3,6,17,17,6,4,28	15 (15)	21 (21)	28 (28)	36 (36)	45 (45)	55 (55)	66 (66)	78 (78)	91 (91)
1	(0,1)	10,27,19,31,20,30,8,6,15,24	15 (15)	21 (21)	28 (28)	36 (36)	45 (45)	55 (55)	66 (66)	78 (78)	91 (91)
1	(0,2)(1,1)(0,1)	16,10,15,29,25,15,9,21	15 (15)	21 (21)	28 (28)	36 (36)	45 (45)	55 (55)	66 (66)	78 (78)	91 (91)
2	(0,1)	2,21,22,3,3,7,13	12 (12)	-	22 (22)	-	35 (35)	-	51 (51)		70 (70)
2	(0,1)	2,31,29,31,14,15,9,3,2,7	15 (12)	-	28(22)	-	45(35)	-	66 (51)	-	91 (70)
2	(0,1)	2,10,29,19,3,17,5,15,17,19,6	15 (15)	-	28 (28)	-	45 (45)	-	66 (66)	-	91 (91)
2	(0,1)(0,1)	2,22,29,25,3,4,31,18	16 (13)	-	29(23)	-	36 (36)	-	67 (52)	-	92 (71)
2	(0,1)(0,1)(0,1)	14,15,14,6,10,31,21,19	21 (15)	-	35 (28)	-	53(45)	-	75 (66)	-	101 (91)

V těchto experimentech bylo dosaženo stejných řešení jako v předchozím případě. U kroku developmentu 2 bylo nalezeno opět kvalitní řešení, jeho princip je ovšem stejný jako na obrázku 17a. Bylo objeveno i několik dalších. Například na obrázku 21a vidíme síť velmi podobnou konvenčně sestrojené síti, první komparátor je však zařazen na konec, tedy do stejné vrstvy jako předchozí komparátor. Tím ušetříme jednu vrstvu síťě, takže při stejném počtu komparátorů dosáneme menšího zpoždění.

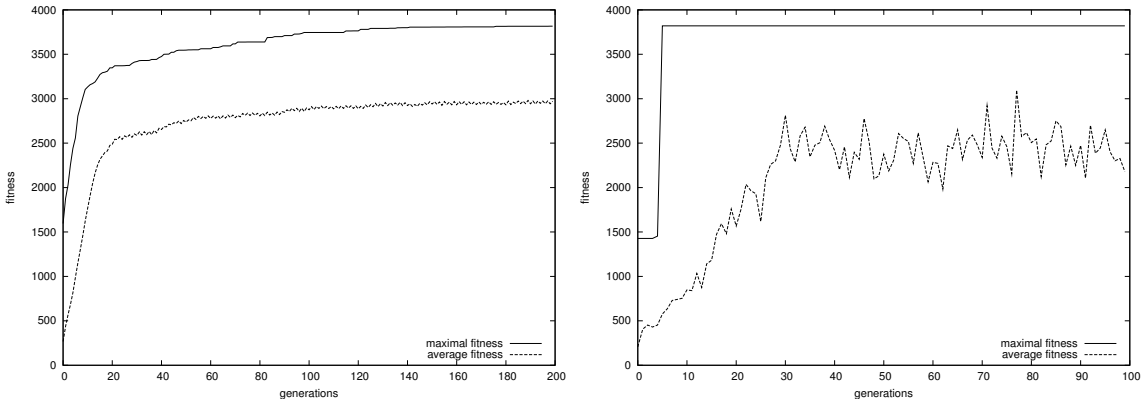
Obrázek 21b ukazuje další princip konstrukce řadicí sítě s využitím jiného embrya vyžadující opět stejný počet komparátorů jako konvenční metoda. Na obrázku 21c je zobrazen příklad sítě zkonstruované s krokem developmentu 2, vyžadující také stejný počet komparátorů jako síť řadící principem vkládání. Obsahuje však redundantní komparátory, které můžeme odstranit a získat tak síť s menším počtem komparátorů než konvenční řešení.



Obrázek 21: Příklady sítí získané z experimentů: (a) $s_d = 1$ s pravidly (19, 10, 17, 17, 10, 8, 7) (b) konstrukce sítě s využitím pravidel (16, 10, 15, 29, 25, 15, 9, 21), (c) síť získaná aplikací pravidel (2, 31, 29, 31, 14, 15, 9, 3, 2, 7) s vyznačenými redundantními komparátory.

Graf na obrázku 22a zobrazuje průměrnou závislot průměrné a maximální fitness. Můžeme všimnout, že křivky jsou strmější než u experimentů, kde bylo embryo pevně dáno. Z toho vyplývá, že zařazení embrya do genotypu zrychluje nalezení kvalitního řešení, ovšem

často „cestou nejmenšího odporu". Tedy zvýhodňuje řešení, která lze nalézt snadněji. Jak vidíme z přehledu v tabulce 6, toto nejsnadnější řešení je založené na dvouvstupém embryu obsahujícím právě jeden komparátor. Toto embryo mělo také nejvyšší úspěšnost při předchozích experimentech. Na obrázku 22b vidíme průběh jednoho cyklu evoluce, kde bylo nalezeno řešení právě s embryem (0,1).



(a) (b)

Obrázek 22: Závislost fitness na počtu generací u experimentů s embryem obsaženým v genotypu a krokem developmentu 1: (a) průměrný průběh, (b) příklad průběhu fitness pro jeden běh algoritmu.

Kapitola 7

Závěr

V této práci jsme si představili konvenční techniky návrhu libovolně velkých řadících sítí a také techniky a algoritmy, které jsme dále využili při jejich evolučním návrhu. Cílem bylo najít způsob, jakým navrhnout pravidla pro konstrukci libovolně velkých řadících sítí, které budou efektivnější z hlediska počtu použitých komparátorů nebo zpoždění než sítě navrhované konvenčním způsobem.

Bylo navrženo několik způsobů evoluce a provedena řada experimentů. První z nich byly cíleny na návrh sítí s libovolným počtem vstupů. Úlohou evoluce bylo nalézt správnou posloupnost pravidel, která umožňovala zkonstruovat libovolně velkou řadicí síť vycházející z konkrétního embrya. Embryí bylo vyzkoušeno několik od dvouvstupého až po pětivstupé. Tyto experimenty ukázaly, že čím je použité embryo menší (čím méně komparátorů obsahuje), tím snadněji se nalezne posloupnost pravidel, která z něj dokáže zkonstruovat libovoně velkou řadicí síť. Naopak, čím je embryo složitější, tím více různých sítí algoritmus dokáže navrhnout, ovšem úspěšnost hledání pravidel je nižší. Celkově s tímto přístupem algoritmus našel s těmito danými pravidly po provedení nastaveného maximálního počtu generací pouze řešení s parametry shodnými s konvenčním přístupem. Kromě znovuobnovení principu vkládání byl nalezen i jiný, z hlediska počtu komparátorů stejný, způsob konstrukce. V průběhu evoluce se objevovala i jiná škálovatelná řešení. Některá z nich obsahovala redundantní komparátory a po jejich odstranění měla výsledná síť menší počet komparátorů, než konvenční řešení.

Pro zlepšení výsledků se začala hledat pravidla, která svou aplikací zvětšila embryo hned o dva vstupy. Tím se snížil počet aplikací pravidel na polovinu pro dosažení stejně velké sítě jako předchozím způsobem konstrukce. Tímto způsobem je však možné konstruovat řadicí sítě pouze o lichém nebo sudém počtu vstupů v závislosti na počtu vstupů embrya. Úspěšnost hledání škálovatelných řešení byla výrazně nižší, zato se podařilo nalézt několik kvalitních řešení. Sítě konstruované nalezenými posloupnostmi pravidel se již vyznačují nižším počtem komparátorů a nižším zpožděním než konvenční řešení. Výhodou nalezeného řešení je pomalejší nárůst počtu komparátorů se zvyšujícím se počtem vstupů. Při počtu vstupů 11 vyžaduje stejný počet komparátorů jako potřebuje konvenčně sestrojená síť pro seřazení deseti vstupů a při počtu vstupů 19 již vyžaduje méně komparátorů než konvenční síť se 17ti vstupy. Rozdíl mezi oběma sítěma o 19 vstupech činí 36 komparátorů. Úspora komparátorů je zejména u vícevstupových sítí relativně velká.

Další modifikace způsobu návrhu pravidel nevyužívala pevně dané embryo. To bylo zahrnuto v genotypu a v průběhu evoluce se vyvíjelo spolu s pravidly. Tento druh experimentu měl určit embryo, které je pro řešenou úlohu nejvhodnější. V případě, že by v předchozích experimentech takové embryo scházelo, mohla tato metoda nalézt ještě lepší řešení. Ovšem

jak se dá odhadnout z výsledků předchozích experimentů, jako nejvýhodnější embryo se ukázalo to nejjednodušší – dvouvstupé embryo obsahující právě jeden komparátor. Opět byly provedeny experimenty pro jeden a dva kroky developmentu. Řešení nalezená touto metodou však nepřinesla nic nového. Úspěšnost, obzvláště s krokem developmentu 2, opět nebyla příliš vysoká. Problém nízké úspěšnosti spočívá nejspíš v pevně dané sadě pravidel, která byla navržena spíše pro tvorbu sítí o libovolném počtu vstupů. Pravděpodobně by bylo výhodnější, kdyby šla pravidla nějakým způsobem parametrizovat.

Literatura

[1] Holland, J. H.: Adaptation in Natural and Artificial Systems. Ann Arbor, University of Michigan Press 1975.

[2] Goldberg D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA, Addison-Wesley 1989.

[3] Michalewicz Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer Verlag 1996..

[4] Koza J. R.: Genetic Programming. Cambridge, MIT Press 1992.

[5] Koza J. R.: Genetic Programming - 2.Cambridge, MIT Press 1994.

[6] Kumar, S., Bentley, J.: On Growth, Form and Computers. Elsevier Academic Press 2003.

[7] Mařík, V., Štěpánková, O., Lažanský, J.: Umělá inteligence (3). Praha, Academia 2000.

[8] Kvasnička, V. a kol.: Evolučné algoritmy, STU Bratislava, 2000.

[9] Foegel, D.B.: Evolutionary Computation. Toward a New Philosophy of Machine Intelligence, IEEE Press, New York, 1995.

[10] Schwefel, H.P.: Numerical Optimization for Computer Models. J. Wiley, Chichester, UK, 1981.

[11] Whitley, D.: A Genetic Algorithm Tutorial. Elektronický dokument dostupný na URL http://samizdat.mines.edu/ga_tutorial/ga_tutorial.ps (leden 2007)

[12] Ryan, C., Collins, J.J., O'Neil, M.: Grammatical Evolution: Evolving Programs for an Arbitrary Language. In EuroGP 1998, str. 83–96.

[13] Sims, K.: Evolving Virtual Creatures. In: Computer Graphics, Annual Conference Series, 1994, str.15-22..

[14] Sekanina, L., Bidlo, M.: Evolutionary Design of Arbitrarily Large Sorting Networks Using Development. Springer Science + Business Media, Inc. 2005.

[15] Knuth, D.: The Art of Computer Programming, Vol. 3 - Sorting and Searching. Addison-Wesley 1973.

[16] Lang, H.W, Flensburg, F.H: Sorting Networks. Elektronický dokument dostupný na URL http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/indexen.htm (prosinec 2006).

Seznam zkratek a symbolů

w	počet vstupů sítě komparátorů
w_e	maximální počet vstupů embrya
f_w	dílčí fitness sítě o w vstupech
s_d	krok developmentu

Přílohy

Na CD jsou přiloženy 2 nástroje použité pro experimenty a zpracování výsledků. Jsou psány v jazyce Haskell pro rodinu POSIXových operačních systému. Byl vyvíjen ve FreeBSD, testován také v OS Linux.

Nástroje

develop

Develop je program, který má na starosti většinu úkonů. Není zamýšlen jako uživatelská aplikace, neobsahuje tedy uživatelské rozhraní, jeho funkce se nastavuje parametry na příkazové řádce. Je to prototyp, který byl sestrojen pouze za účelem experimentů, proto ani nebyl kladen důraz na jeho optimalizaci.

Použití

Program přijímá následující parametry na příkazové řádce.

- *-b, –batchmode* (bez parametrů) – dávkový režim. V tomto režimu nezobrazí výsledek v grafické podobě a nečeká na reakci uživatele.
- *-d, –develop {seznam pravidel}* – zkonstruuje řadicí sítě až do počtu vstupů 20 z daného embrya za použití seznamu pravidel, který je předán jako parametr. Seznam pravidel je ve tvaru [n,n, ..., n], kde n značí číslo pravidla. U každé sítě vypíše, zda je planou řadicí sítí a počet komparátorů a počet použitých komparátorů (bez redundantních). Na výpisu 1 vidíme výstup programu v tomto režimu.
- *-e, –embryo {embryo}* – nastaví embryo, které se má použít při evoluci, developmentu, nebo vizualizaci. Emryo je ve tvaru $[(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$, kde x a y jsou jednotlivé složky komparátoru popsané v odstavci 3.1.
- *-g, –generations {počet generací}* – specifikuje počet generací v evolučním procesu. Pokud není zadáno, nastaví se výchozí hodnota 200 generací.
- *-l, –logfile {název souboru}* – specifikace souboru, do kterého se uloží výsledek evolučního návrhu. Název souboru se udává včetně cesty, bez cesty se uloží do aktuálního adresáře.
- *-r, –remove_ redundand* (bez parametru) – při vizualizaci se nezobrazují redundantní komparátory.

- *-s, –step {krok developmentu}* – nastaví krok developmentu s_d. Povolené hodnoty jsou pouze 1 a 2.
- *-v, –visualize {seznam pravidel}* – vytvoří a zobrazí embryo a 3 vývojové kroky sítě podle seznamu pravidel daném v parametru. Formát je stejný jako u parametru -d.
- *-V, –visualizelog {název souboru}* – režim vizualizace uložených výsledků.

Režimy činnosti

Bez jakýchkoli parametrů program spustí jeden *evoluční proces*. Každou generaci vypisuje informace o nejlepším jedinci. Po dokončení určeného počtu generací zobrazí výsledek na terminál a otevře okno s grafickým znázorněním konstrukce sítě. V sítích vyznačí červeně redundantní komparátory.

```
# ./develop
Evolving with embryo: [4] [(0,1),(2,1),(1,2),(0,2),(1,1)]
1) 1356, 00110100010110001011000010110011000010011001101000110010,  [26,22,22,11,24,6,20,18]
2) 3739, 00001010011101010100101010100001110010100100100110000100,  [5,29,9,10,14,9,19,4]
3) 3806, 00111000101010000110100010110011000010011001101000110010,  [28,10,13,11,24,6,20,18]
...
199) 3796, 01111110110001011100000010100010001010111000011100101011,  [31,17,24,10,17,14,14,11]
200) 3796, 01111110110001011100000010100010001010111000011100101011,  [31,17,24,10,17,14,14,11]
embryo: [(0,1),(2,1),(1,2),(0,2),(1,1)]
Scalable: True
```

Výpis 1: Zkrácený výstup z programu develop spuštěného bez parametrů

Režim *konstrukce sítě*. Parametr -d postupně konstruuje sítě a ověřuje jejich platnost až do počtu vstupů 20. Zobrazuje délku sítě v komparátorech a délku bez redundantních vstupů. Výstup vidíme na výpisu 2.

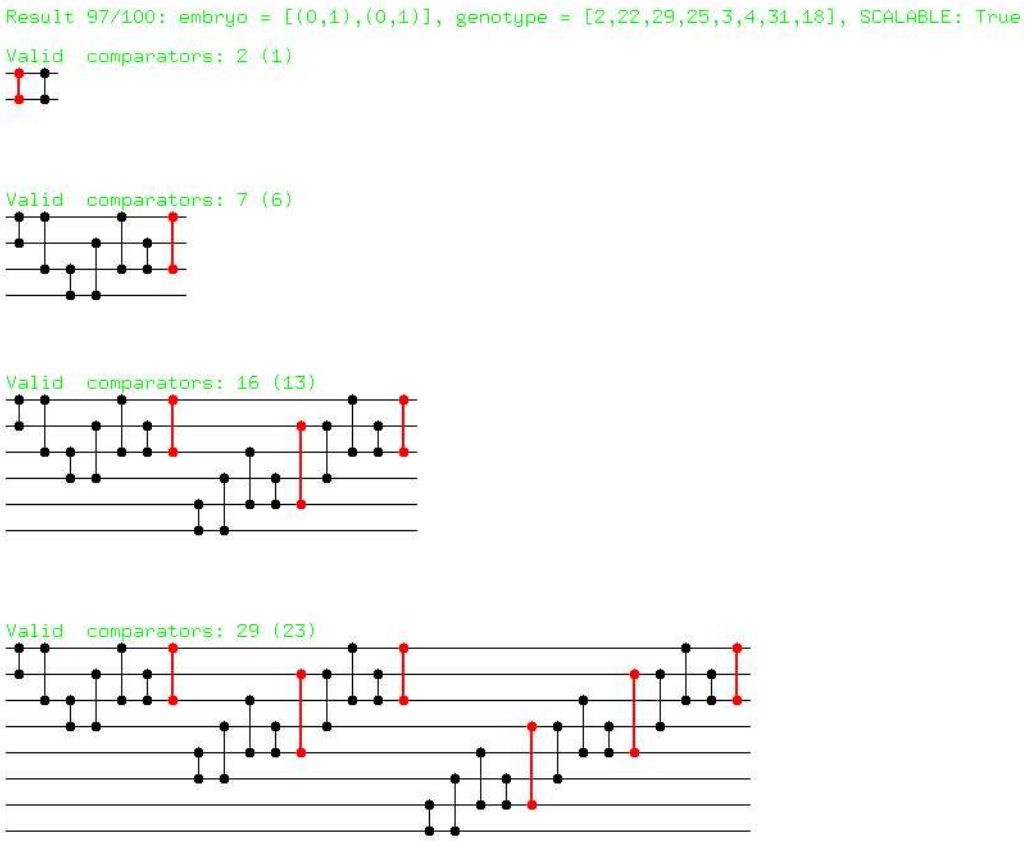
```
Inputs: 6       Length: 20 (14)         Valid
Inputs: 7       Length: 33 (20)         Valid
Inputs: 8       Length: 47 (27)         Valid
Inputs: 9       Length: 64 (35)         Valid
Inputs: 10      Length: 82 (44)         Valid
Inputs: 11      Length: 103 (54)        Valid
Inputs: 12      Length: 126 (65)        Valid
Inputs: 13      Length: 151 (77)        Valid
Inputs: 14      Length: 178 (90)        Valid
Inputs: 15      Length: 207 (104)       Valid
Inputs: 16      Length: 238 (119)       Valid
Inputs: 17      Length: 271 (135)       Valid
Inputs: 18      Length: 306 (152)       Valid
Inputs: 19      Length: 343 (170)       Valid
Inputs: 20      Length: 382 (189)       Valid
```

Výpis 2: Výstup z programu develop spuštěného s parametry: -d "[28,0,29,11,10,19,26,9]" -e "[(0,3),(1,3),(0,2),(1,2),(

Režim *vizualizace výsledků*. Přečte soubor s uloženými údaji a umožňuje procházet všechny výsledky. Každý výsledek zobrazí jako postupnou konstrukci sítě s vyznačením redundantních komparátorů (pokud není program spuštěn s parametrem -r, který způsobí

jejich úplné vynechání). Screenshot vidíme na obrázku 1. U každé sítě je zobrazena informace o počtu komparátorů všetně redundandních i bez nich a její platnosti. V případě, že je platná je text vybarven zeleně, jinak červeně.

Mezi jednotlivými výsledky se přesouvá levou a pravou šipkou. Vizualizace se ukončuje dvojitým stiskem klávesy Esc.



Obrázek 1: Snímek obrazovky vizualizačního režimu (barvy upraveny kvůli tisku).

parseStats

Program zpracovává statistiky o průběhu fitness, které se ukládají do adresáře *statistics*. Generuje příkazy pro *gnuplot*[1], které způsobí vygenerování grafu.

Jeho použití je prosté:

```
./parseStats statistics/statistic_e0 | gnuplot
```

Graf je uložen ve formátu *eps* a má stejné jméno jako soubor se statistikami s přidanou příponou *.eps*. Prohlédnout si jej můžeme například programem *gv*:

```
gv statistics/statistics_e0.eps
```

[1] http://www.gnuplot.org/

Do adresáře *statistics* se automaticky ukládá průběh evoluce. Soubor je označen velikostí embrya obsaženého v genotypu. Každý další běh programu *develop* data do souboru přidává. Program generující graf z těchto statistik půměruje všechy statistiky, které v daném souboru jsou. Chceme-li zobrazit statistiku pouze jednoho běhu, je potřeba před spuštěním evolučního procesu soubor smazat.

Adresářová struktura

- *doc/* – obsahuje tuto dokumentaci.
- *experiments/* – obshuje dříve prováděné experimenty, vizualizovatelné programem develop spuštěným s parametrem -V.
- *src/* – všechny zdrojové kódy.
- *statistics/* – adresář, kam se automaticky ukládají statistiky o průběhu evoluce.
- *./* – kořenový adresář obsahující spustitelné soubory a Makefile.

Kompilace

K programu je přiložen Makefile, za předpokladu, že je nainstalován kompilátor ghc verze 6.6 nebo 6.4, stačí v adresáři se souborem Makefile spustit příkaz *make*. Spustitelný soubor se jmenuje *develop*.

V případě, že z jakéhokoli důvodu není možné make použít, je možné program zkompilovat příkazem

```
ghc --make -O2 Main -o develop,
```

což je stejný příkaz, který je zadán v Makefile.

Použití v prostředí CVT na FIT VUT v Brně

Protože ve školním prostředí (konkrétně na počítači merlin) je stará verze ghc (6.4), která neobsahuje ve standardních knihovnách knihovny *ByteString*, přiložil jsem ji ke zdrojovým kódům, tak aby bylo možné program bez problému zkompilovat i na této verzi překladače.

Na počítači *merlin* je nainstalováno vše potřebné a je dostupný všem, postup budeme tedy vysvětlovat na něm.

Kompilace

Kompilace se nijak neliší od té, která byla popsaná v sekci 3. Pro správné zkompilování programu stačí v kořenovém adresáři develop (není třeba přecházet do adresáře *src,* je jedno, kde zda se přikaz spustí v adresáři *develop* či *src*) spustit příkaz make. Na výpisu 3 vidíme postup přihlášení na počítač merlin a kompilace programu za předpokladu, že leží v adresáři ~login/develop/, kde login je přihlašovací jméno uživatele.

```
ssh login@merlin.fit.vutbr.cz
cd develop
make
```

Spouštění

Máme několik možností. Nejjednodušší je z libovolné linuxové stanice zkompilovat a v grafickém prostředí spustit.

```
cd develop
make
./develop [parametry]
```

Program lze spustit na jakékoli linuxové stanici s X11. Pokud na dané stanici není překladač ghc, je možné program zkompilovat vzdáleně na počítači merlin a poté zkopírovat k sobě.

```
ssh -X login@merlin.fit.vutbr.cz make -C develop
scp -r login@merlin.fit.vutbr.cz develop .
./develop [parametry]
```

Další možností je přímo vzdálené spouštění z počítače merlin příkazem:

```
ssh -X login@merlin.fit.vutbr.cz make -C develop
ssh -X login@merlin.fit.vutbr.cz develop/develop [parametry]
```

Programová dokumentace

Jak bylo zmíněno výše, nástroje jsou je psány v jazyce Haskell. Programová dokumentace se týká programu *develop*, protože nástroj parseScript obsahuje čtyři krátké funkce, které jsou komentovány přímo ve zdrojovém kódu.

Program *develop* se sesává z několika modulů, z nichž každý je v souboru stejného názvu, pouze s příponou .hs, případně v podadresářích podle hierarchie modulů.

Main

Vstup do programu, obstarává zpracování parametrů z příkazové řádky a spouší požadované akce. Popis parametrů, které přijímá, a jejich význam naleznete v uživatelské příručce.

Simulator

Modul starající se o ověřování správnosti řadicích sítí, odstraňování redundantních komparátorů a generování vstupů pro kontrolu správnosti řazení.

Seznam exportovaných funkcí

```
isValid         :: [[Int]]      -> [(Int, Int)] -> Bool
removeRedundant :: Int          -> [(Int, Int)] -> IO [(Int, Int)]
correctlySorted :: [[Int]]      -> [(Int, Int)] -> Int
generateInputs  :: Int          -> [[Int]]
inputCount      :: [(Int, Int)] -> Int
```

- *isValid* – vrátí True, pokud síť zadaná druhým parametrem seřadí správně všechny vstupní posloupnosti dané prvním parametrem.

- *removeRedundant* – odstraní redundandní komparátory ze sítě zadané druhým parametrem. První parametr udává počet vstupů sítě.
- *correctlySorted* – vrátí počet správně seřazených posloupností dané prvním parametrem řadicí sítí dané parametrem druhým.
- *generateInputs* – vygeneruje všech 2^n kombinací řetězců z abecedy {0,1} délky n, kde n značí počet vstupů sítě a je dán prvním parametrem.
- *inputCount* – vrátí index nejvyššího vstupu sítě, který je dána prvním parametrem.

Visualize

Modul zajišťující vizualizaci výsledků.

Seznam exportovaných funkcí

```
displayLog :: Bool -> Int -> String -> IO ()
display    :: Bool -> Int -> Int    -> [(Int, Int)]
                   -> [Int] -> String -> IO ()
```

- *displayLog* – spustí režim vizualizace výsledků. První parametr určuje, zda mají zobrazovat redundantní komparátory, druhý krok developmentu s_d a třetí název souboru, ve kterém jsou výsledky uloženy.
- *display* – zobrazí konstrukci konkrétní sítě. První parametr udává, zda se mají zobrazovat redundantní komparátory druhý určuje krok developmentu, třetí počet vstupů první sítě konstruované z embrya, dalším parametrem je embryo, následuje seznam pravidel (genotyp) a nakonec text, který se do okna vypíše (informace o škálovatelnosti apod.)

Evolution.Chromosome

Modul obsahující funkce pro manipulaci s chromozomy.

Seznam exportovaných datových typů

```
Chromosome {
    fitness     :: Int,
    geneLen     :: Int,
    embryoLen   :: Int,
    ew          :: Int,
    genes       :: B.ByteString,
    genotypeLen :: Int,
    bounds      :: (Int, Int)}
```

- *Chromosome* postupně obsahuje hodnotu fitness, informaci o délce genu (počet bitů tvořící 1 gen), délku embrya obsaženého v genotypu (počet komparátorů), maximální počet vstupů embrya, geny (řetězec znaků 0 a 1), délku genotypu (počet genů), rozmezí, ve kterém se hodnoty genů (pravidel) smí pohybovat (min, max).
- *CrossOver* – typ křížení, může nabývat hodnot CrossOnePoint, CrossTwoPoint nebo CrossUniform.

Seznam exportovaných funkcí

```
fromList         :: Int -> Int -> [(Int,Int)] -> [Int]-> Chromosome
toList           :: Chromosome -> ([Int], [(Int, Int)])
randomChromosome :: Int -> Int -> (Int, Int) -> IO Chromosome
crossOver        :: Chromosome -> Chromosome
                    -> CrossOver -> IO (Chromosome, Chromosome)
mutate           :: Int -> Chromosome -> Chromosome
fillFitness      :: (Chromosome -> IO Int) -> Chromosome -> IO Chromosome
```

- *fromList* – vyrobí chromozom ze seznamu tvořící embryo (3. parametr) a posloupnost čísel pravidel (4. parametr). První dva parametry udávají minimální a maximální hodnotu, jakou může číslo pravidla nabývat.
- *toList* – převede zadaný chromozom na seznam pravidel a embryo.
- *randomChromosome* – vytvoří náhodný chromozom. První parametr udává délku embrya, druhý počet pravidel v chromozomu. Ve třetím parametru je uložena minimální a maximální hodnota čísla pravidla.
- *crossOver* – provede křížení dvou zadaných chromozomů, daného typu (3. parametr).
- *mutate* – zmutuje zadaný chromozóm, prvním parametrem je míra mutace, což je číslo od 0 do 1000, které odpovídá pravděpodobnosti mutace 0-1%.
- *fillFitness* – pomocná funkce pro vyplnění fitness v chromozomu. Fitness fukce se předá jako první parametr, chromozom jako druhý, vrátí chromozom s vyplněnou fitness.

Evolution.Development

Modul zajišťuje vývoj sítě z embrya aplikací pravidel v chromozomu.

Seznam exportovaných funkcí

```
development :: Int -> Int -> Int -> [Int] -> [(Int, Int)]
               -> IO [(Int, Int)]
maxRules    :: Int
defaultDevelopmentStep :: Int
```

- *develpment* – sestrojí síť z embrya daného posledním parametrem za pomocí pravidel daných předposledním parametrem o počtu vstupů daném třetím parametrem. První síť vytvořená z embrya má počet vstupů daný druhým parametrema a první parametr je krok developmentu.
- *maxRules* – konstatní funkce vracející počet používaných pravidel včetně NOP.
- *defaultDevelopmentStep* – výchozí krok developmentu (jeho hodnota se použije, není-li krok developmentu na příkazové řádce definován)

Evolution.Genetic

Modul implementující genetický algoritmus.

Seznam exportovaných datových typů

```
Parameters {
    chromosomeLen    :: Int,
    chromosomeBounds :: (Int, Int),
    crossOverType    :: CrossOver,
    mutationFactor   :: Int,
    populationSize   :: Int,
    maxGenerations   :: Int,
    embryoLength     :: Int,
    shouldExit       :: (Int -> Bool),
    fitnessFunct     :: (Chromosome -> IO Int),
    afterGeneration  :: (Int -> Chromosome -> IO ())}
```

Struktura obsahuje informace o nastavení evolučního procesu. Atributy po sobě znamenají délku chromozomu, rozmezí, ve kterém se mohou hodnoty genů pohybovat, faktor mutace, velikost populace, maximální počet generací, délku embrya obsaženého v chromosomu, funkci, které se po každé generaci předá fitness nejlepšího jedince a pokud vrátí True, evoluční proces se ukončí. Standardně nastavena na konstatní funkci vracející False. Dále obsahuje fitness funkci a nakonec funkci, která se provádí po každé generaci. jejími parametry je číslo generace a nejlepší jedinec. Slouží například pro vypisování průběhu evoluce.

Seznam exportovaných funkcí

```
evolve :: Parameters -> IO [Chromosome]
```

- *evolve* – spustí evoluční výpočet podle zadaných parametrů. Po ukončení vrátí populaci sestupně seřazenou podle fitness.

Evolution.Utils

Seznam exportovaných funkcí

```
binStrToInt     :: B.ByteString -> Int
intToBinStr     :: Int -> B.ByteString
alignToLength   :: Int -> B.ByteString -> B.ByteString
optimize        :: [(Int, Int)] -> [(Int, Int)]
```

- *binStrToInt* – převede zadaný binární řetězec na odpovídající číslo
- *intToBinStr* – převede zadané číslo na odpová
- *alignToLength* – zarovná binární řetězec do určité minimální délky. Je-li např. „101”, pak jeho zarovnání na délku 5 vypadá „00101”.
- *optimize* – odstraní ze sítě duplikované komparátory a komparátory, které mají připojeny oba vstupy na stejný vstup sítě.